Langage de description d'attaques pour la détection d'intrusions par corrélation d'événements ou d'alertes en environnement réseau hétérogène

Cédric Michel

N° d'ordre: 2943

THESE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Cédric Michel

Equipe d'accueil : SSIR, Supélec (Campus de Rennes)
Ecole Doctorale : MATISSE
Composante universitaire : IFSIC

Titre de la thèse :

Langage de description d'attaques pour la détection d'intrusions par corrélation d'événements ou d'alertes en environnement réseau hétérogène

soutenue le 16 Décembre 2003 devant la commission d'examen

M. :	Jean-Marc	JEZEQUEL	Président
MM. :	Frédéric	Cuppens	Rapporteurs
	Michel	Dupuy	
MM. :	Gérardo	Rubino	Examinateurs
	Ludovic	Mé	

à Emmanuelle et Adrien,
à mes parents.

Remerciements

Je remercie Jean-Marc Jézéquel, professeur à l'université de Rennes 1, qui m'a fait l'honneur de présider ce jury de thèse.

Je remercie Frédéric Cuppens, professeur à l'ENST Bretagne, et Michel Dupuy, chef du CERTA, d'avoir bien voulu juger mon travail de recherche en acceptant la charge de rapporteur.

Je remercie Gérardo Rubino, directeur de recherche à l'INRIA, d'avoir accepté de diriger cette thèse.

Je ne remercierai jamais assez Ludovic Mé, enseignant-chercheur à Supélec dans l'équipe Sécurité des Systèmes et Réseaux (SSIR), pour avoir co-dirigé cette thèse et l'avoir encadrée au quotidien. Je lui suis reconnaissant de m'avoir donné ma chance. Je le remercie également pour sa disponibilité et pour tous les conseils qu'il m'a prodigués tout au long de ma thèse.

Je remercie tous les enseignants-chercheurs de l'équipe SSIR pour leur accueil chaleureux. Je tiens à remercier Eric Totel et Bernard Vivinis à qui j'ai mené la vie dure durant les phases de relecture de ce mémoire, Laurent Heye pour son aide durant tout le projet MIRADOR, ainsi que Christophe Bidan, Véronique Alanou et Bernard Jouga pour leurs conseils et leur disponibilité.

Je remercie la Direction Générale de l'Armement (et notamment le service DGA/CELAR/CASSI) pour avoir initié le Programme d'Etude Amont MIRADOR qui a permis de financer cette thèse. Je tiens à remercier tous les intervenants de ce projet (Alcatel-CIT, Onéra Chatillon, Onéra Toulouse, ENST Bretagne et Supélec) pour leur collaboration et pour m'avoir fait apprécier le travail en équipe.

Je remercie tous mes collègues doctorants/docteurs (Zakia Marrakchi, Jacob Zimmermann, Benjamin Morin, Nicolas Prigent, Thomas Duval et tous les autres) pour tous les échanges intéressants que nous avons eus (et pour avoir ri à certaines de mes blagues !).

Je dédie cette thèse à ma femme Emmanuelle et à mon fils Adrien. Merci de m'avoir supporté (dans tous les sens du terme) pendant ces années.

Résumé

La détection d'intrusions vise à automatiser la détection des actions malicieuses perpétrées sur un réseau de machines par un utilisateur interne ou un attaquant externe. Cela passe par la mise en place d'une surveillance des activités des utilisateurs et des systèmes pour analyse ultérieure de ces activités.

Nous proposons dans cette thèse un langage de haut niveau d'abstraction, ADeLe, dédié à la description des attaques. Ce langage donne les moyens de décrire complètement une attaque sous ses différents aspects : exploit, détection et réaction.

Nous détaillons tout d'abord la syntaxe et la sémantique informelle de ce langage. La partie détection de la description permet de corréler des événements ou des alertes en définissant des signatures qui comportent des contraintes temporelles et logiques.

Nous donnons ensuite la sémantique opérationnelle associée à la partie détection du langage, basée sur des automates à états finis qui modélisent les signatures. Nous présentons un algorithme abstrait qui utilise ces automates pour effectuer la détection. Nous abordons également le problème de l'explosion combinatoire lié à notre modèle qui conserve en cours d'exécution, par défaut, l'ensemble des solutions partielles en mémoire.

Nous présentons enfin différentes réalisations logicielles : plusieurs capteurs système et réseau, une sonde applicative, un compilateur du langage ADeLe vers nos automates à états finis ainsi que l'analyseur ADeLaIDS. Nous présentons également deux expérimentations effectuées à l'aide de cet analyseur.

Mots clés : sécurité des systèmes d'information, détection d'intrusions, langage de description d'attaques, approche par scénarios, signature d'attaque, corrélation d'événements, corrélation d'alertes, automates de reconnaissance.

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

L'approche traditionnelle de la sécurité des systèmes d'information (SSI) consiste à mettre en place des mécanismes de sécurité préventifs pour assurer le respect d'une politique de sécurité visant à garantir la confidentialité, l'intégrité ou la disponibilité de ressources sensibles.

Mettre en place des mécanismes préventifs est une condition nécessaire mais pas suffisante. En effet, compromettre un système est essentiellement une question de temps et de moyens mis en œuvre. Malgré tous les efforts déployés pour renforcer la sécurité d'un système ou d'un réseau, tout mécanisme de sécurité a ses limites et peut être contourné. Par exemple :

- un pare-feu, aussi sophistiqué soit-il, ne peut filtrer que les paquets qui le traversent. Il sera donc inopérant si un utilisateur le contourne en utilisant un modem sur une machine du réseau interne ;
- un contrôle d'accès à base de login/mot de passe n'empêche pas la connexion d'une personne non autorisée si elle a obtenu frauduleusement le mot de passe d'un utilisateur légitime ;
- un protocole ou un mécanisme bien pensé et qu'on ne peut théoriquement pas détourner n'est pas à l'abri d'une erreur de programmation lors de sa mise en œuvre. Ainsi de nombreux logiciels que l'on croyait sûrs se révèlent vulnérables à des attaques de type *buffer overflow*[1] qui permettent généralement à l'attaquant d'exécuter le code de son choix avec des privilèges plus élevés que prévu.

Ainsi, malgré toute la sécurité préventive mise en place, une intrusion peut quand même survenir. Il est alors primordial de savoir que cela s'est produit : pour comprendre comment, pour empêcher que cela recommence et pour réparer les dégâts éventuels. Il faut donc mettre en place une surveillance des activités (sur les systèmes et les réseaux qui les interconnectent) afin d'y rechercher de manière automatique ce qui pourrait constituer une intrusion.

C'est le rôle de la détection d'intrusions de détecter les violations de la politique de sécurité, qu'elles correspondent au contournement d'un mécanisme de

[1] débordement de données dans une zone tampon

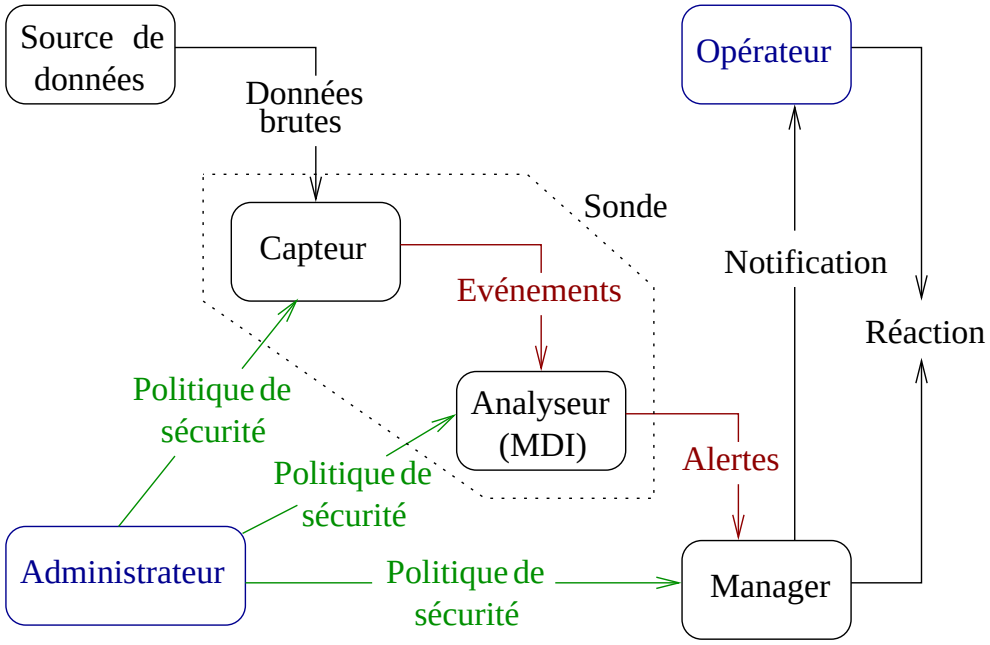

FIG. 1 – Modèle générique de la détection d'intrusions proposé par l'IDWG

sécurité préventif ou à l'abus par un utilisateur légitime des pouvoirs qui lui ont été accordés.

Pour détecter des intrusions, deux approches principales ont été proposées. La première, dite approche comportementale, consiste à modéliser dans une phase initiale le comportement normal d'entités du système pour rechercher ensuite des attitudes déviantes dans le comportement courant de ces entités. La seconde, dite approche par scénarios, consiste à rechercher dans les activités des entités des traces de scénarios d'attaque connus, exploitant les nombreuses vulnérabilités présentes dans les systèmes utilisés aujourd'hui.

L'IDWG (*Intrusion Detection Working Group*) de l'IETF a défini dans [WE02] un modèle générique de la détection d'intrusions qui représente les fonctionnalités communes à tous les IDS (qu'ils utilisent l'approche comportementale ou l'approche par scénarios).

La figure 1 reproduit ce modèle et permet d'introduire un certain nombre de concepts. Le lexique situé en annexe A présente les définitions des termes apparaissant dans la figure ainsi que d'autres couramment utilisés dans ce mémoire.

Dans la figure 1, on peut voir le processus complet de la détection avec le cheminement des données au sein du système. L'administrateur configure les différents composants (capteur(s), analyseur(s), manager(s)). Les capteurs accèdent aux données brutes, les filtrent et les formatent pour ne renvoyer que les événements intéressants à un analyseur. Les analyseurs utilisent ces événements pour décider de la présence ou non d'une intrusion et envoient le cas échéant une alerte au manager (qui notifie l'opérateur humain). Une réaction éventuelle peut être menée automatiquement par le manager ou manuellement par l'opérateur.

Un IDS particulier n'est pas forcément organisé comme dans la figure 1. Il peut regrouper plusieurs modules logiques en un seul module logiciel ou bien comporter plusieurs instances d'un type de module (ex : plusieurs capteurs ou plusieurs analyseurs).

On note que ce modèle est récursif par nature. En effet, on ne précise pas sur le schéma d'où proviennent les données brutes (qui peuvent donc être des données de plus haut-niveau telles que les alertes). Dans ce cas précis, un manager peut contenir un module permettant de faire de la corrélation à partir de plusieurs alertes (considérées alors comme des événements de base), et à son tour émettre de nouvelles alertes.

Problématique et objectifs de la thèse

Nos travaux de recherche s'inscrivent dans le contexte des systèmes de détection d'intrusions utilisant l'approche par scénarios. Cette approche impose de construire une base contenant toutes les signatures d'attaque que l'on souhaite détecter dans le flux des activités surveillées. Chaque signature définit quelles sont les activités caractéristiques générées par l'attaque qui lui est associée.

Ecrire une signature d'attaque demande une connaissance approfondie de tous les aspects liés à cette attaque : les vulnérabilités qu'elle exploite, son mode opératoire, ses conséquences, les traces caractéristiques qui permettent de la détecter et les mesures éventuelles à prendre après détection.

Malheureusement, la dispersion des informations disponibles publiquement sur une attaque sont à l'origine d'un gaspillage conséquent du temps et des moyens consacrés à la détection d'intrusions. En effet, chaque administrateur de sécurité et chaque chercheur, lorsqu'il doit se documenter sur une attaque, refait le même travail de collecte et de vérification des informations. Ce travail est nécessaire mais ne devrait être effectué qu'une seule fois. Le problème est qu'il n'existe pas de langage permettant d'exprimer et de rassembler toute cette expertise de manière satisfaisante. On peut considérer que la base de signatures de l'IDS réseau Snort[2]

[2]http://www.snort.org

joue ce rôle fédérateur au sein de la communauté de la détection d'intrusions, mais il est limité à un seul aspect de l'attaque : la signature.

Un autre problème se pose. La plupart des IDS actuels ne permettent d'exprimer que des signatures relativement simples. Par exemple, l'IDS réseau Snort fonde son diagnostic de détection sur l'analyse d'un seul paquet. Cette granularité insuffisante dans l'expression des signatures est à l'origine d'un taux important de fausses alertes car, pour ne pas rater la détection d'une attaque, on risque d'écrire une signature approximative (qui sera un sur-ensemble de ce que l'on veut détecter). Cela implique également que l'on ne peut pas exprimer de manière satisfaisante la détection des attaques qui comportent plusieurs étapes.

L'objectif de cette thèse est de combler ces lacunes en proposant un langage de haut niveau d'abstraction, ADeLe[3], qui permet d'exprimer le plus complètement possible les différents aspects liés à une attaque au sein d'une seule description. Ce langage est dédié à la détection d'intrusions et permet l'écriture de signatures complexes (qui permettent la corrélation entre plusieurs événements ou alertes).

Un tel langage ne doit pas se restreindre à la seule spécification de signatures pour détecter l'attaque. Il doit permettre d'exprimer et de lier entre eux les multiples aspects d'une attaque.

Le premier aspect concerne la mise en œuvre de l'attaque vue du côté de l'attaquant. Il fournit un résumé condensé des pré-requis et des conséquences de l'attaque, comme les privilèges initiaux que doit posséder l'attaquant pour réussir l'attaque et les gains de privilèges qu'il obtient en cas de succès. Il peut également contenir le mode opératoire de l'attaque, c'est-à-dire la description des actions enchaînées par l'attaquant. On peut alors mettre en œuvre cette attaque facilement (manuellement ou par le biais d'un interpréteur/compilateur adapté au langage utilisé dans la description) afin d'évaluer la capacité d'un IDS donné à détecter l'attaque décrite.

Le deuxième aspect concerne la détection de l'attaque, vue du défenseur. Plusieurs descriptions complémentaires sont nécessaires pour configurer des systèmes de détection d'intrusions basés sur l'approche par scénarios. Il s'agit des descriptions :

- de la signature caractéristique de l'attaque (qui permettra la détection proprement dite),
- de la vérification des conséquences d'une attaque détectée (afin d'affiner le diagnostic et de préciser si l'attaque a réussi ou echoué),
- de l'alerte qui sera créée et envoyée à destination de l'administrateur.

Pour la signature, nous pouvons notamment préciser des contraintes sur l'ordre

[3] **A**ttack **De**scription **L**anguag**e**

dans lequel les événements doivent être observés. Nous ne faisons toutefois pas d'hypothèse sur les algorithmes qui seront utilisés pour le vérifier. C'est un choix qui permet de maintenir une certaine séparation entre le langage et les algorithmes qui seront utilisés par les différents IDS. Ainsi, nous n'utilisons pas d'opérateurs qui seraient trop spécifiques à un IDS particulier. Cela empêcherait la traduction d'une description écrite en ADeLe vers d'autres IDS ne comportant pas ces opérateurs.

Le dernier aspect concerne la réaction à l'attaque. Il donne la possibilité d'exprimer les éventuelles contre-mesures qui peuvent être prises automatiquement en cas de détection de cette attaque (pour la stopper ou corriger ses effets). Cette caractéristique doit être utilisée avec prudence car elle peut facilement être détournée de son objectif initial par un attaquant pour provoquer des dénis de service.

Disposer d'un tel langage de description d'attaques permet de fédérer non seulement les efforts de la communauté de la détection d'intrusions mais également ceux de la communauté des CERT. En effet, l'exploitation d'une nouvelle vulnérabilité peut être exprimée dans un langage commun, ce qui favorise à la fois la diffusion et la compréhension de la description correspondante.

Les caractéristiques du langage

Nous mettons ici en avant les caractéristiques dont nous avons souhaité doter le langage ADeLe et qui ont guidé sa conception.

Description complète

Le but premier d'ADeLe est de donner la possibilité de représenter chacun des aspects d'une attaque au sein d'une seule description, c'est-à-dire :

- **du point de vue de l'attaquant** : on peut souvent trouver de l'information brute sur les attaques comme, par exemple, le code source d'un exploit. Afin de savoir comment s'effectue réellement l'attaque, nous pouvons inclure ce code dans la description. Il peut être écrit dans les langages traditionnels utilisés pour les exploits (C, Perl, NASL ...) ou dans un autre langage de haut niveau que nous proposons dans 2.2.2. Mais disposer du code source ne suffit pas : nous pouvons également décrire les préconditions qui font que l'attaque est possible (OS, version d'application vulnérable, type d'accès initial, services réseaux disponibles ...) ainsi que les conséquences de l'attaque (informations obtenues, gains de privilèges, ressources utilisées, dénis de service ...).
- **du point de vue du défenseur** : nous pouvons préciser quels sont les événements caractéristiques observables qui doivent se produire durant l'at-

taque. La description contient aussi les relations temporelles et contextuelles entre ces événements qui permettront de détecter l'attaque. Nous pouvons également exprimer ce qui permet de confirmer la détection en vérifiant la présence des conséquences attendues de l'attaque pour préciser si la tentative a abouti. Chaque description contient également un modèle précisant quelles informations devront figurer dans l'instance de l'alerte remontée après la détection. Dans certains cas, une réaction appropriée et automatique à l'attaque détectée est souhaitable et peut être spécifiée dans la description.

Expressivité

Le langage ADeLe a été conçu pour être le moins restrictif possible quant aux possibilités de description d'attaques. Comme l'information de base pour la détection est l'événement, une description doit donc pouvoir utiliser les événements provenant de tous les types de capteurs : système, réseau et applicatif. De même, si l'on souhaite faire de la corrélation entre plusieurs alertes (pour en générer d'autres de plus haut niveau), il faut pouvoir utiliser les alertes provenant des IDS.

Pour atteindre ce double objectif, la représentation de ces événements et alertes doit être relativement homogène : c'est pourquoi nous préconisons l'utilisation de formats utilisant le langage XML[Wor00]. Nous recommandons pour les alertes le format IDMEF (qui est en passe de devenir un standard) et pour les événements un format proche de l'IDMEF (mais adapté à chaque source de données) que nous définissons en annexe E. Ainsi, parler d'une valeur contenue dans une occurence d'événement ou d'alerte revient simplement à désigner un champ du document XML correspondant.

Une description en ADeLe doit également pouvoir modéliser la détection d'attaques complexes, c'est à dire composées de plusieurs étapes. Cela signifie mettre en relation plusieurs événements ou alertes pour en déduire la présence d'une attaque. Nous avons donc défini dans la syntaxe d'ADeLe des opérateurs de corrélation temporelle et contextuelle entre les événements/alertes, qui permettent respectivement de donner des contraintes sur leur enchaînement et sur les valeurs qu'ils contiennent.

Modularité

Une description en ADeLe se doit d'être modulaire afin que l'on puisse réutiliser au mieux les descriptions déjà écrites. Cela vaut à la fois pour la partie attaque et pour la partie détection : on peut donc d'abord définir des briques de base, puis les réutiliser pour composer des descriptions de plus haut niveau de manière hiérarchique.

Lorsque l'on définit une attaque de haut niveau qui résulte de la composition

de plusieurs attaques déjà décrites, on dispose alors du mode opératoire de chacune des étapes. Il est donc inutile de répéter à nouveau toutes ces informations dans la nouvelle attaque : il suffit d'invoquer le nom des attaques élémentaires (éventuellement avec des paramètres) pour disposer du code associé. C'est à peu près analogue à regrouper l'appel en séquence de plusieurs programmes dans un script shell. Les éventuels paramètres permettent d'enchaîner les étapes : les sorties (résultats) d'une attaque élémentaire peuvent servir d'entrée à l'attaque suivante.

Il en va de même pour la partie détection. Si l'on veut corréler plusieurs alertes provenant d'IDS pour détecter une attaque de plus-haut niveau, on utilise simplement dans le scénario les noms des alertes élémentaires et on décrit leur enchaînement temporel ainsi que les contraintes qui les relient (ex : même source, même cible ...).

Généricité

De nouvelles variantes d'attaques connues apparaissent tous les jours. Une modification même mineure peut permettre à l'attaque de passer inaperçue dans un premier temps puisque les IDS utilisant une base de signatures ne la connaitront pas encore. C'est le problème majeur qu'on prête à l'approche par scénarios : être obligé de connaître toutes les variantes d'une attaque afin de pouvoir les détecter[4]. Cela signifierait dans notre contexte être obligé de faire autant de descriptions distinctes qu'il existe de variantes.

Nous apportons une réponse partielle à ce problème dans ADeLe en introduisant des opérateurs pour exprimer et factoriser plusieurs variantes en une seule description. Ainsi, il est fréquent que l'ordre de certaines étapes d'une attaque composée soit indifférent et amène au même résultat. Une autre possibilité est qu'une des étapes est interchangeable avec une autre produisant les mêmes effets. Nos opérateurs permettent justement de représenter non-déterminisme et alternative à la fois dans la partie attaque et dans la partie détection d'une description. L'introduction de la généricité permet donc de réduire la taille de la base de description d'attaques puisqu'une seule description peut représenter plusieurs variantes de la même attaque.

Aptitude à tester des IDS

Comme une description en ADeLe peut contenir le code source de l'attaque, constituer une base de descriptions d'attaques signifie également constituer une base d'attaques exécutables. Moyennant l'utilisation du compilateur (ou de l'interpréteur) approprié, il est possible, dans un réseau de tests confiné, de jouer l'intégralité de cette base d'attaques contre des systèmes surveillés par des IDS.

[4]le même problème se pose pour les logiciels antivirus.

Cela permet de tester l'efficacité de la détection de ces IDS, notamment en termes de *faux négatifs*. En effet, puisque l'on sait quelles attaques ont été réellement jouées, il est possible de dire pour chacune d'entre elles si l'IDS considéré l'a effectivement détectée.

On peut évidemment se poser le problème de la mise à disposition publique d'une telle base d'attaques, même restreinte à la communauté de la détection d'intrusions. Il nous semble que ce problème, même s'il doit être débattu, ne doit pas entraver le développement d'un outil permettant de tester les IDS.

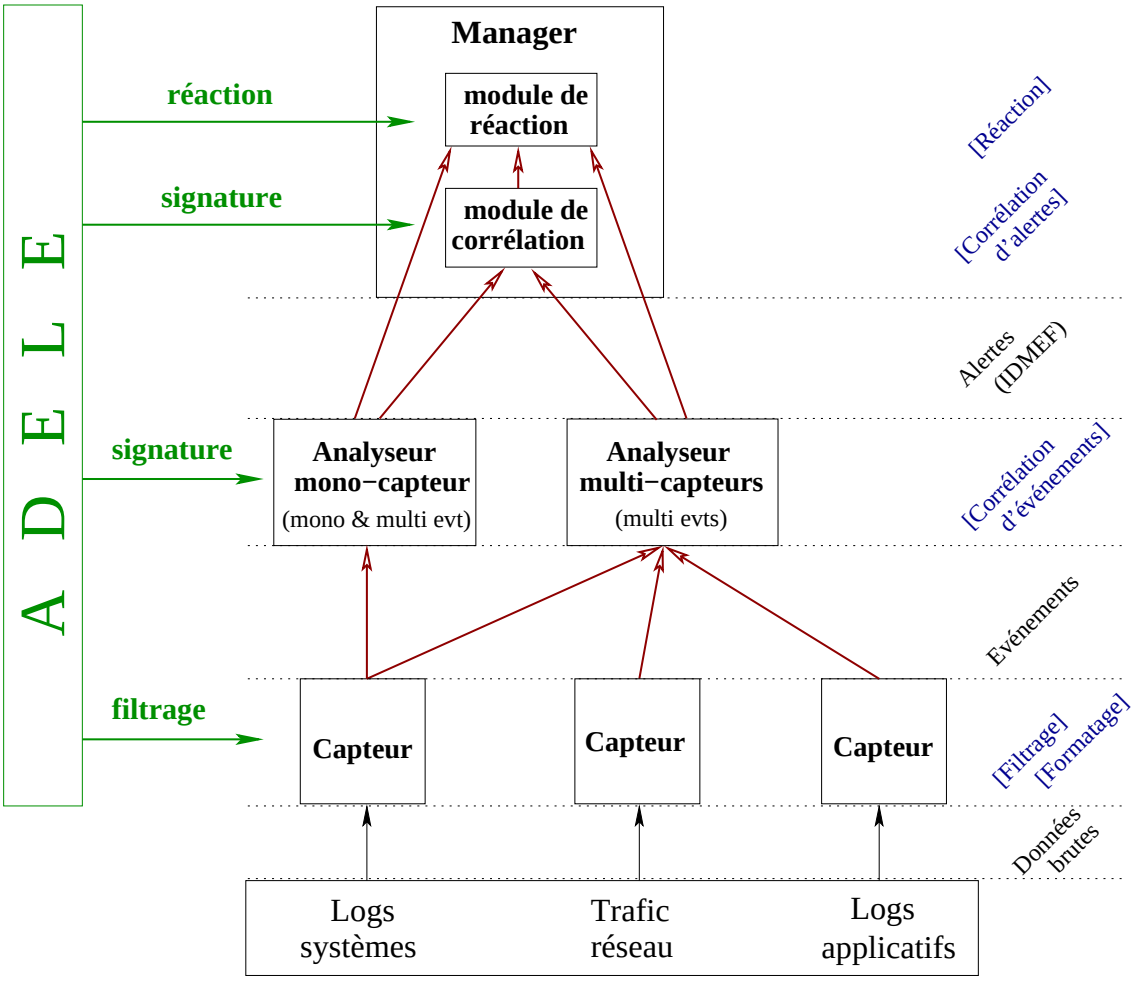

FIG. 2 – Configuration de la détection à partir d'ADeLe

Aptitude à configurer des IDS
Une description d'attaque en ADeLe doit permettre de configurer des IDS, c'est-à-dire des capteurs et les analyseurs associés. Toutefois, une description en ADeLe n'est pas directement opérationnelle. Certes, elle contient de l'information sur une attaque mais cette information a besoin d'être extraite et transformée par des compilateurs pour être utilisable concrètement.

La figure 2 représente l'objectif visé de configuration des IDS à partir du langage ADeLe. Elle montre, dans une architecture standard, les différents modules pouvant être configurés.

Une **signature** basée sur une corrélation d'événements peut être fournie à un analyseur. La liste des événements attendus dans la signature peut alors être utilisée pour réaliser un **filtrage** au niveau des capteurs (afin qu'ils n'émettent que ces types d'événements).

Si la **signature** est basée sur une corrélation d'alertes, elle peut être fournie à un module de corrélation au niveau du manager.

Si la description d'attaque comporte un descriptif de la **réaction** à entreprendre après détection, il peut être fourni à un module de réaction du manager.

Il faut donc écrire un compilateur adapté à chacun des IDS (capteur et analyseur). Au cas où la description en ADeLe dépasse les possibilités de l'analyseur cible, il faudra peut-être se restreindre à traduire et utiliser seulement un sous-ensemble du langage ADeLe. C'est le cas, par exemple, avec la nouvelle version de G$^{\text{A}}_{\text{S}}$SA$_{\text{T}}$A , baptisée G$^{\text{N}}$G [5], que nous avons utilisée au cours du projet Mirador pour valider expérimentalement le concept de configuration d'IDS.

Guide de lecture

Ce mémoire est organisé de la manière suivante.

Dans le chapitre 1, nous dressons un état de l'art du domaine de la détection d'intrusions. Tout d'abord, nous présentons une classification des systèmes de détection d'intrusions. Nous passons ensuite en revue les différentes catégories de langages utilisées jusqu'à présent pour décrire les différents aspects d'une attaque.

Dans le chapitre 2, nous donnons une spécification de la syntaxe du langage de description d'attaques ADeLe. Nous présentons en détail les différentes parties d'une description en ADeLe qui correspondent à chacun des aspects de l'attaque : exploit, détection et réponse.

[5] qui signifie G$^{\text{A}}_{\text{S}}$SA$_{\text{T}}$A New Generation

Dans le chapitre 3, nous traitons de la sémantique opérationnelle associée à la partie détection du langage ADeLe.

Dans le chapitre 4, nous présentons quelques réalisations : la programmation de quelques capteurs, l'écriture d'un compilateur du langage ainsi que la mise en oeuvre de l'analyseur ADeLaIDS associé au langage ADeLe.

Finalement, nous concluons par une réflexion sur des extensions à apporter au langage ADeLe ainsi que des axes de recherche pour prolonger ce travail.

Les annexes contiennent quelques compléments techniques : grammaires, DTD, algorithme de détection, exemples de descriptions d'attaque...

Chapitre 1

Etat de l'art

Dans ce chapitre, nous présentons un bref état de l'art du domaine de la détection d'intrusions.

Dans la section 1.1, nous dressons un bref panorama de la détection d'intrusions en réalisant une classification des systèmes de détection d'intrusions (IDS).

Pour décrire une attaque aussi précisément que possible, il faut considérer plusieurs aspects dont chacun comporte des informations de nature différente. Une description d'attaque nécessite donc l'utilisation de plusieurs langages dédiés. Dans la section 1.2, nous passons en revue les différents types de langages utilisés jusqu'à présent en référençant des travaux qui s'y rapportent.

Dans la section 1.3, nous présentons le positionnement dans la classification présentée d'un IDS configurable à partir du langage ADeLe.

1.1 Une classification des systèmes de détection d'intrusions

Le nombre de systèmes de détection d'intrusions existants ou ayant existé étant important (environ 130 répertoriés dans [MH03] en janvier 2003), nous ne visons pas ici à l'exhaustivité. Nous exposons seulement leurs caractéristiques communes.

Nous présentons une classification selon différents critères qui ne sont pas forcément mutuellement exclusifs : par exemple, un système de détection d'intrusions (IDS) peut mettre en œuvre deux analyseurs utilisant des méthodes différentes.

La classification adoptée n'est pas hiérarchique : elle présente tour à tour et au même niveau les catégories caractérisant chaque IDS. Elle utilise les critères suivants (cf figure 1.1) :

- la source des données à analyser,
- la méthode de détection utilisée,

- le lieu de l'analyse des données,
- la fréquence de l'analyse,
- le comportement en cas d'attaque détectée.

Notre classification est voisine de celle qui a été proposée dans [DDW00]. Nous considérons un autre critère qui est celui de la réaction à adopter après détection. Par contre, nous ne prenons pas en compte le critère du «paradigme de détection» qui constitue un sur-ensemble du critère lié à la méthode de détection utilisée.

1.1.1 Source des données à analyser

Les sources possibles de données à analyser sont une caractéristique essentielle des systèmes de détection d'intrusions puisque ces données constituent la matière première du processus de détection. Les données proviennent soit de logs générés par le système d'exploitation, soit de logs applicatifs, soit d'informations provenant du réseau, soit encore d'alertes générées par d'autres IDS.

Source d'information système

Un système d'exploitation fournit généralement plusieurs sources d'information :

- **commandes systèmes** : presque tous les systèmes d'exploitation fournissent des commandes pour avoir un «instantané» de ce qui se passe. Ainsi, sous UNIX, des commandes telles que *ps* ou *vmstat* fournissent des informations précises sur les activités courantes du système.
- **accounting** : l'accounting fournit de l'information sur l'usage des ressources partagées par les utilisateurs (temps processeur, mémoire, espace disque, débit réseau, applications lancées, ...). Les modules statistiques et neuronaux d'Hyperview [DBS92] ont utilisé cette source d'information.
- **audit de sécurité** : tous les systèmes d'exploitation modernes proposent ce service pour fournir des événements système, les associer à des utilisateurs et assurer leur collecte dans un fichier d'audit. On peut donc potentiellement disposer d'informations sur tout ce que font (ou ont fait) les utilisateurs : accès en lecture à un fichier, exécution d'une application, etc. Par exemple, l'audit BSM[Sun] représente les appels systèmes produits par les programmes qui s'exécutent sur un système Solaris.

Les outils utilisant cette source de données sont appelés *Host Based Intrusion Detection Systems* (HIDS).

Source d'information réseau

Des dispositifs matériels ou logiciels (sniffers) permettent de capturer le trafic

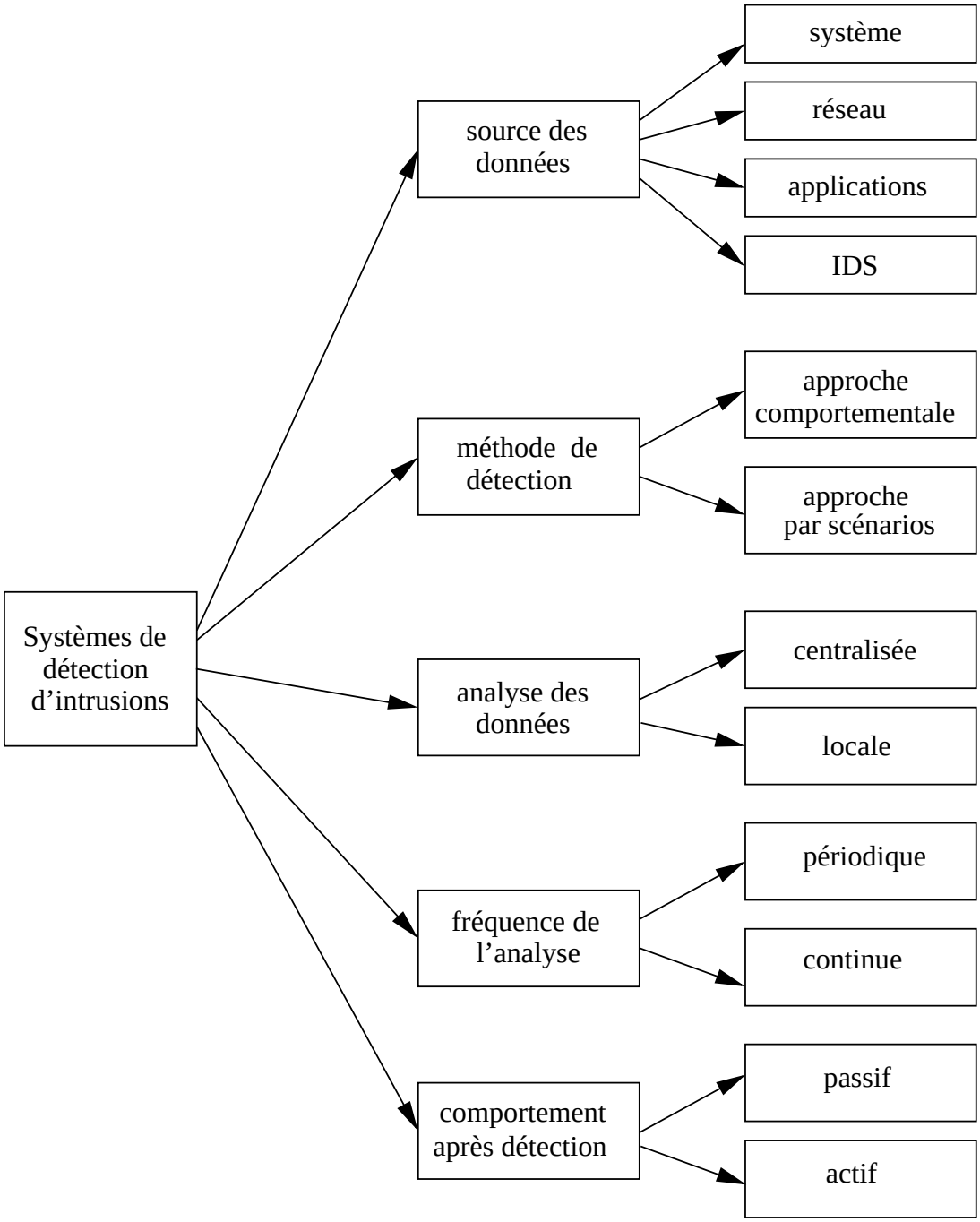

FIG. 1.1 – Caractéristiques des systèmes de détection d'intrusions

réseau. Cette source d'information est particulièrement adaptée lorsqu'il s'agit de rechercher les attaques en déni de service qui se passent au niveau réseau ou les tentatives de pénétration à distance. Le processus d'interception des paquets peut être rendu quasiment invisible pour l'attaquant car on peut utiliser une machine dédiée juste reliée à un brin du réseau, configurée pour ne répondre à aucune sollicitation extérieure et dont personne ne soupçonnera l'existence. Néanmoins, il est difficile de garantir l'origine réelle de l'attaque que l'on a détectée car il est facile de masquer son identité en modifiant les paquets réseau. De plus, l'utilisation du chiffrement peut rendre impossible la détection si elle basée sur le contenu utile des paquets (les données brutes sans les entêtes).

Les trappes SNMP[1] constituent également une source d'information réseau. Elle est particulièrement adaptée à la collecte des informations provenant des équipements matériels (routeurs, switches, ...) car ils supportent presque toujours le protocole SNMP.

Presque tous les outils commerciaux récents utilisent les informations issues du réseau. Les outils utilisant cette source de données sont appelés *Network-based Intrusion Detection Systems* (NIDS).

Source d'information applicative

Les applications peuvent également constituer une source d'information pour les IDS [Sie99]. Les capteurs applicatifs sont de deux natures :

- **capteur interne** : le filtrage sur les activités de l'application est alors exécuté par le code de l'application [AL01][2].
- **capteur externe** : le filtrage se fait à l'extérieur de l'application. Plusieurs méthodes sont utilisées : un processus externe peut filtrer les logs produits par l'application [ADD00] ou bien l'éxécution de l'application peut être interceptée (au niveau de ses appels de librairies [LJ01] ou d'un proxy applicatif (ex : Netsecure Web [Cal03])).

Prendre ses informations directement au niveau de l'application présente plusieurs avantages. Premièrement, les données interceptées ont réellement été reçues par l'application. Il est donc difficile d'introduire une désynchronisation entre ce que voit passer le capteur applicatif et ce que reçoit l'application contrairement à ce qu'il peut se passer avec les capteurs réseau [PN98]. Ensuite, cette source d'information est généralement de plus haut niveau que les sources système et réseau. Cela permet donc de filtrer des événements qui ont une sémantique plus riche. Finalement, si l'on prend l'exemple d'une connection web chiffrée par SSL,

[1] les agents SNMP (Simple Network Management Protocol) envoient des notifications d'événements, appelées trappes, à un système de gestion.

[2] dans ce cas précis, l'extraction est faite dans le code de l'application et les données sont filtrées par un processus externe.

un capteur réseau ne verra passer que des données pseudo-aléatoires tandis qu'un capteur associé au serveur web pourra analyser le texte en clair de la requête.

Cette source de données reste aujourd'hui encore assez peu utilisée par les outils commerciaux et les prototypes de recherche.

Source d'information basée IDS

Une autre source d'information, souvent de plus haut niveau que les précédentes, peut être exploitée. Il s'agit des alertes remontées par des analyseurs provenant d'un IDS. Chaque alerte synthétise déjà un ou plusieurs événements intéressants du point de vue de la sécurité. Elles peuvent être utilisées par un IDS pour déclencher une analyse plus fine à la suite d'une indication d'attaque potentielle. De surcroît, en corrélant plusieurs alertes, on peut parfois détecter une intrusion complexe de plus haut niveau. Il y aura alors génération d'une nouvelle alerte plus synthétique que l'on qualifie de méta-alerte.

La frontière entre un événement et une alerte est parfois floue car tout dépend à quel niveau d'abstraction on se place. C'est le cas notamment chez les IDS où le capteur et l'analyseur sont indissociables. Ces IDS détectent généralement des attaques simples qui ne requièrent qu'un seul événement. Nous les appelons IDS mono-événement (ex : diagnostic de remontée d'alerte à partir d'un seul paquet réseau). Le filtrage sur les données brutes et la détection se font alors en une seule phase. Il n'y a pas par contre pas d'ambiguïté dans le cas des IDS qui détectent des attaques plus complexes, à partir de plusieurs événements (nous les appellons IDS multi-événements). Parmi les rares IDS multi-événements qui opèrent une détection à partir d'alertes, on peut citer Emerald [VS01], AlertSTAT [RSG02] ainsi que les trois IDS du projet Mirador[3] (les modules CRIM [CM02], CRS et la dernière version de GASSATA).

Cette source de données commence à être exploitée depuis peu (les premières publications datent de l'année 2000).

1.1.2 Méthodes de détection

Les deux approches[4] (permettant de classer les méthodes de détection) proposées à ce jour sont l'**approche comportementale** (*anomaly detection*) et l'**approche par scénarios** (*misuse detection* ou *knowledge-based detection*). La première se base sur l'hypothèse que l'on peut définir un comportement «normal» de l'entité à surveiller (utilisateur, service, application ...) et que toute déviation par rapport à celui-ci est potentiellement suspecte. La seconde s'appuie sur la connaissance des techniques employées par les attaquants : on en tire des scéna-

[3]projet financé par le DGA/CELAR/CASSI qui dépend du Ministère de la Défense français.

rios d'attaque et on recherche dans les traces d'audit leur éventuelle occurence.

L'approche comportementale

Le comportement normal (profil) d'un utilisateur, d'une application ou d'un système peut être construit de différentes manières. Le système de détection d'intrusions compare l'activité courante à ce profil. Tout comportement déviant est alors considéré intrusif. Cela permet potentiellement de détecter des attaques inconnues auparavant.

On peut distinguer deux catégories de profils :

1. **profils construits par apprentissage (empiriques)**

   Parmi les méthodes proposées pour construire les profils par apprentissage, les plus marquantes sont les suivantes :

   - **méthode statistique** : le profil est calculé à partir de variables considérées comme aléatoires et échantillonnées à intervalles réguliers. Ces variables peuvent être le temps processeur utilisé, la durée et l'heure des connexions, etc. Un modèle statistique (ex : covariance) est alors utilisé pour construire la distribution de chaque variable et pour mesurer, au travers d'une grandeur synthétique, le taux de déviation entre un comportement courant et le comportement passé. L'outil phare des années 1987-1991, NIDES [JVL+93], utilise cette méthode.
   - **système expert** : ici, c'est une base de règles qui décrit statistiquement le profil de l'utilisateur au vu de ses précédentes activités. Son comportement courant est comparé aux règles, à la recherche d'une

[4]Une autre manière de classifier, reprenant la dichotomie «comportemental / scénarios», peut être retenue. Il s'agit de la dichotomie «*state-based* / *transition-based*» introduite dans [DDW00]. L'intérêt de ce nouveau découpage est sa plus grande généralité. En considérant que le système surveillé passe, grâce à des transitions, par différents états jusqu'à un état attaqué, la détection peut s'opérer à deux niveaux différents.

Dans l'approche *transition-based*, on choisit de surveiller les transitions pour caractériser une attaque, il faut donc connaître complètement son mode opératoire et ses éventuelles variantes. Par exemple, si une certaine suite de commandes est reconnue, on détecte l'attaque correspondante.

Par contre, dans l'approche *state-based*, on choisit de surveiller l'état du système, c'est-à-dire de l'évaluer régulièrement et de décider selon des critères prédéfinis si l'on est en présence d'une attaque. Par exemple, si la signature MD5 d'un exécutable a changé depuis le dernier test, on peut craindre la présence d'un cheval de Troie (sans même savoir comment l'intrus s'y est pris).

Pour revenir à la classification usuelle, l'approche par scénarios correspond exactement à l'approche *transition-based*. L'approche comportementale est, quant à elle, un sous-ensemble de l'approche *state-based* qui englobe également l'utilisation d'outils de contrôle statique tels que SAINT ou Tripwire. Cette dernière manière de détecter des intrusions est parfois appelée «statique» dans le sens où il y a contrôle statique périodique de l'état du système. On détecte alors une attaque par ses conséquences, que l'on espère limitées. Ces outils de contrôle statique ne constituent cependant pas des systèmes de détection d'intrusions stricto sensu.

anomalie. La base de règles est rafraîchie régulièrement. L'outil Wisdom & Sense [VL89] utilise cette méthode, aujourd'hui tombée en désuétude.

- **réseaux de neurones** : la technique consiste à apprendre à un réseau de neurones le comportement de l'entité à surveiller. Par la suite, lorsqu'on lui fournira en entrée les actions courantes effectuées par l'entité, il devra décider de leur normalité. L'outil Hyperview[5] [DBS92] comporte un module de ce type.
- **analyse de signatures** : l'approche proposée par Forrest [FHS97] s'inspire de l'immunologie biologique et met en œuvre des algorithmes de *pattern matching*. Il s'agit de construire un modèle de comportement normal des services réseaux Unix. Le modèle consiste en un ensemble de courtes séquences d'appels système représentatifs de l'exécution normale du service considéré. Des séquences d'appels étrangères à cet ensemble sont alors considérées comme l'exploitation potentielle d'une faille du service.

Pour toutes ces méthodes, le comportement de référence utilisé pour l'apprentissage étant rarement exhaustif, on s'expose à des risques de fausses alarmes (faux positifs). De plus, si des attaques ont été commises durant cette phase, elles seront considérées comme normales (risque de faux négatifs).

2. **profils spécifiant une politique de sécurité (*policy-based*)**

   Pour les IDS dits *policy-based*, il n'y a pas de phase d'apprentissage. Leur comportement de référence est spécifié par une politique de sécurité : la détection d'une intrusion intervient chaque fois que la politique est violée.

   Historiquement, Ko, Fink et Levitt ont proposé dans [**?**] un IDS système dit *specification-based*. Le profil est ici une politique de sécurité qui décrit la suite des appels systèmes licites d'une application (sous Unix). Cette approche ressemble aux travaux de Forrest [FHS97] mais le profil décrit le comportement légal attendu et non un comportement observé empiriquement.

   Ensuite, d'autres approches, utilisant la surveillance des appels systèmes sous Linux, ont été proposées. Dans [KR02], Ko et Redmond présentent un IDS système où le profil est une politique de sécurité qui précise quel groupe d'utilisateurs est autorisé à interférer avec quel groupe de données. Dans [ZMB02], Zimmermann, Mé et Bidan proposent un IDS système où le profil est une politique de sécurité représentée par un graphe qui définit

[5] pour être exact, Hyperview base son modèle sur la prédiction d'une action future (même si l'apprentissage se fait sur des actions passées).

les flux d'informations autorisés entre objets du système. Dans ces deux cas, toute séquence d'appels systèmes, pour être conforme à la politique de sécurité (profil), doit vérifier un certain prédicat (théorème prouvé). Par opposition, une suite d'appels systèmes qui ne vérifie pas ce prédicat contient une violation de la politique de sécurité.

L'approche par scénarios

On construit des scénarios d'attaque en spécifiant ce qui est caractéristique de l'attaque et qui doit être observé dans les traces d'audit. L'analyse des traces d'audit se fait à la recherche de ces scénarios. Les méthodes proposées à ce jour sont les suivantes :

- **système expert** : le système expert comporte une base de règles qui décrit les attaques. Les événements d'audit sont traduits en des faits qui sont interprétables par le système expert. Son moteur d'inférence décide alors si une attaque répertoriée s'est ou non produite. Cette méthode a été utilisée dans [LJ88] et dans ASAX [HCMM92]. Excepté l'outil EMERALD (qui intègre le système expert P-Best [**?**]), les outils récents ne l'utilisent plus.
- **analyse de signatures** : il s'agit là de la méthode la plus en vue actuellement. Des signatures d'attaques sont fournies à des niveaux sémantiques divers selon les outils (de la suite d'appels système aux commandes passées par l'utilisateur en passant par les paquets réseau). Divers algorithmes sont utilisés pour localiser ces signatures connues (voir [DDMW98] pour un exemple) dans les traces d'audit. Ces signatures sont toujours exprimées sous une forme proche des traces d'audit. Si l'on prend l'exemple des NIDS, les algorithmes de recherche de motifs utilisés permettent d'obtenir de bonnes performances en vitesse de traitement mais sont générateurs de nombreuses fausses alertes. En effet, les signatures que l'on peut écrire (par exemple avec Snort [Roe99]) sont simplistes et permettent difficilement de corréler plusieurs paquets entre eux. La plupart des IDS commerciaux (tels que Realsecure [Sys98] ou NetRanger [Cis98]) utilisent cette méthode.
- **automates à états finis** : plusieurs IDS utilisent des automates à états finis pour coder le scénario de reconnaissance de l'attaque. Cela permet d'exprimer des signatures complexes et comportant plusieurs étapes. On passe d'un état initial sûr à un état final attaqué via des états intermédiaires. Chaque transition entre états est déclenché par des conditions sur les événements remontés par les capteurs. L'outil IDIOT [KS94] utilise des réseaux de Pétri colorés (*Colored Petri Nets*) pour modéliser les scénarios d'attaque, tandis que l'outil NetSTAT [VK99] utilise des diagrammes de transitions d'états (*State Transition Diagrams*) pour les représenter.

L'approche par scénarios permet de savoir précisément quelle attaque a été détectée. Néammoins, elle ne permet de détecter que des attaques connues ce qui oblige à maintenir à jour la base de scénarios d'attaque.

1.1.3 Localisation de l'analyse des données

On peut également faire une distinction entre les IDS en se basant sur la localisation réelle de l'analyse des données :

- **analyse centralisée** : certains IDS ont une architecture multi-capteurs (ou multi-sondes). Ils centralisent les événements (ou alertes) pour analyse au sein d'une seule machine. L'intérêt principal de cette architecture est de faciliter la corrélation entre événements puisqu'on dispose alors d'une vision globale. Par contre, la charge des calculs (effectués sur le système central) ainsi que la charge réseau (due à la collecte des événements ou des alertes) peuvent être lourdes et risquent de constituer un goulet d'étranglement.
- **analyse locale** : si l'analyse du flot d'événements est effectuée au plus près de la source de données (généralement en local sur chaque machine disposant d'un capteur), on minimise le trafic réseau et chaque analyseur séparé dispose de la même puisance de calcul. En contrepartie, il est impossible de croiser des événements qui sont traités séparément et l'on risque de passer à côté de certaines attaques distribuées.

1.1.4 Fréquence de l'analyse

Une autre caractéristique des systèmes de détection d'intrusions est leur fréquence d'utilisation :

- **périodique** : certains systèmes de détection d'intrusions analysent périodiquement les fichiers d'audit à la recherche d'une éventuelle intrusion ou anomalie passée. Cela peut être suffisant dans des contextes peu sensibles (on fera alors une analyse journalière, par exemple).
- **continue** : la plupart des systèmes de détection d'intrusions récents effectue leur analyse des fichiers d'audit ou des paquets réseau de manière continue afin de proposer une détection en quasi temps-réel. Cela est nécessaire dans des contextes sensibles (confidentialité) et/ou commerciaux (confidentialité, disponibilité). C'est toutefois un processus coûteux en temps de calcul car il faut analyser à la volée tout ce qui se passe sur le système.

1.1.5 Comportement après détection

Une autre façon de classer les systèmes de détection d'intrusions consiste à les classer par type de réaction lorsqu'une attaque est détectée :

- **passive** : la plupart des systèmes de détection d'intrusions n'apportent qu'une réponse passive à l'intrusion. Lorsqu'une attaque est détectée, ils génèrent une alarme et notifient l'administrateur système par e-mail, message dans une console, voire même par beeper. C'est alors lui qui devra prendre les mesures qui s'imposent.
- **active** : d'autres systèmes de détection d'intrusions peuvent, en plus de la notification à l'opérateur, prendre automatiquement des mesures pour stopper l'attaque en cours. Par exemple, ils peuvent couper les connexions suspectes ou même, pour une attaque externe, reconfigurer le pare-feu pour qu'il refuse tout ce qui vient du site incriminé. Des outils tels que RealSecure [Sys98] ou NetProwler[6] proposent ce type de réaction. Toutefois, il apparait que ce type de fonctionnalité automatique est potentiellement dangereux car il peut mener à des dénis de service provoqués par l'IDS. Un attaquant déterminé peut, par exemple, tromper l'IDS en usurpant des adresses du réseau local qui seront alors considérées comme la source de l'attaque par l'IDS. Il est préférable de proposer une réaction facultative à un opérateur humain (qui prend la décision finale).

1.2 Les langages de description d'attaques

L'approche par scénarios est fondée sur la connaissance du mode opératoire de l'attaque et, par conséquent, sur la connaissance des activités caractéristiques de cette attaque (observables dans les systèmes ou les réseaux surveillés).

Pour décrire complètement une attaque, il faut décrire de nombreux aspects de l'attaque qui sont par nature complètement différents. Par conséquent, si l'on veut décrire une attaque sous toutes ses faces, il faut utiliser plusieurs langages adaptés. Dans [VEK00, EVK00], Vigna, Kemmerer et Eckmann donnent une très bonne classification de ces langages. Six classes distinctes sont définies :

- les langages d'**exploit** servent à décrire le mode opératoire utilisé pour effectuer une intrusion ;
- les langages d'**événements** décrivent le format utilisé pour représenter les événements ;
- les langages de **détection** sont utilisés pour décrire les manifestations d'une attaque dans les activités des systèmes ou des réseaux ;

[6] http://www.symantec.fr/region/fr/product/netprowler.html

– les langages de **corrélation d'alertes** sont destinés à permettre l'analyse des alertes fournies par les IDS afin de générer des méta-alertes ;
– les langages de **description d'alertes** décrivent le format utilisé pour construire l'alerte qui sera remontée après détection ;
– les langages de **réaction** sont utilisés pour exprimer les éventuelles contremesures à prendre après détection.

Nous abordons chacun de ces types de langages dans les sections suivantes.

1.2.1 Les langages d'exploit

Les langages d'exploit sont utilisés pour décrire les étapes à suivre pour perpétrer une intrusion. Ce sont généralement des langages à usage général (tels que C, C++, Perl, Bourne Shell...) mais il existe également des langages conçus spécialement pour l'écriture d'attaques ou de tests de vulnérabilités . On peut citer CASL (Custom Attack Simulation Language) [Sec98] et NASL (Nessus Attack Scripting Language) [Der99].

Il y a un autre aspect qui n'est pas cité dans [VEK00, EVK00] et qui, selon nous, a sa place dans les langages d'exploit : ce sont les pré-conditions et post-conditions d'une attaque. Il nous semble important de pouvoir également préciser les pré-conditions nécessaires pour que l'attaque ait lieu (OS, vulnérabilités...) ainsi que les post-conditions ou conséquences de l'attaque (gains de privilèges, obtention d'informations...). C'est d'autant plus important si l'on souhaite réutiliser la description pour rejouer l'attaque afin de tester des IDS. A notre connaissance, les langages qui traitent ces deux aspects sont : LAMBDA [CO00], ADeLe [MM01] et IDLE [LMPT98].

1.2.2 Les langages d'événements

Les langages d'événements sont utilisés pour la description des événements. Les événements sont une représentation des données brutes filtrées et constituent la matière première du processus de détection.

Beaucoup de langages d'événements sont spécialisés et adaptés à un seul type d'événement. Ainsi, NADF est le format des événements utilisés par ASAX [HCMM92]. Le format Tcpdump [JLM00] est souvent utilisé pour la capture des paquets réseau. De même, le format BSM [Sun] est utilisé pour les traces de l'audit système Solaris.

D'autres langages sont plus généralistes et permettent de décrire des événements de nature différente. Le format Syslog [Sys] est utilisé dans les systèmes Unix pour enregistrer des messages systèmes et applicatifs. Bishop [Bis95] a proposé un format standard pour les événements : chaque enregistrement est consti-

tué de marqueurs qui séparent des suites '*champ=valeur*'. Dans la même lignée que le format de Bishop, mais en plus structuré, on peut utiliser le langage XML pour représenter des événements.

1.2.3 Les langages de détection

Les langages de détection permettent d'exprimer comment on peut reconnaître les manifestations d'une attaque. C'est dans ces langages que l'on va codifier la connaissance que l'on a de l'attaque, c'est-à-dire sa signature. De nombreux langages ont été proposés, avec presque autant de mécanismes différents pour implémenter la détection. Presque tous ces langages sont dédiés à un IDS unique.

Le langage de l'IDS réseau Bro [Pax98] est un vrai langage de programmation proche du C (où les signatures sont des fonctions que l'on doit programmer).

Le langage RUSSEL est le langage propriétaire utilisé par l'IDS système ASAX [Mou97] dont les signatures sont exprimées sous forme de règles.

Le langage STATL [EVK00] utilisé par les IDS de la suite logicielle STAT (USTAT, NetSTAT...) permet d'écrire des signatures sous formes d'automates en donnant explicitement la liste des états et des transitions ainsi que les traitements associés à chaque état.

Le langage des signatures de l'IDS réseau Snort [Roe99] permet d'exprimer des règles. Chaque règle est associée à une attaque connue et décrit les caractéristiques que doit posséder un paquet réseau pour détecter la signature.

Les signatures exprimées dans le langage Sutekh [PD00] sont des expressions combinant des filtres d'événements à l'aide d'opérateurs de séquence et d'ordre partiel. Chaque filtre contient des contraintes sur les valeurs des champs des événements. La sémantique de Sutekh est basée sur des règles de réécriture.

Les signatures exprimées dans le langage de Logweaver [**?**] sont des expressions combinant des filtres d'événements à l'aide d'opérateurs de séquence, d'ordre partiel et de disjonction. La sémantique de Logweaver est basée sur des automates à états finis dont les transitions sont étiquetées par les filtres.

1.2.4 Les langages de corrélation d'alertes

De la même façon qu'on se base sur des événements pour obtenir des alertes, les langages de corrélation d'alertes permettent de spécifier comment on peut obtenir des méta-alertes[7] à partir d'alertes. Certaines attaques ne sont que des étapes intermédiaires d'une attaque de plus grande envergure. Un manager peut ainsi utiliser des alertes comme source de données pour obtenir des méta-alertes (alertes de plus haut-niveau).

[7] la distinction entre alertes et méta-alertes n'intervient qu'au niveau conceptuel car les deux utilisent le même format

A notre connaissance, ADeLe [MM01], LAMBDA [CO00], JIGSAW [TL00] et CRS[MD03] semblent être les seules références de **langages** qui permettent de corréler des alertes pour détecter des attaques de plus haut-niveau. Même si dans Emerald [VS01] on a affaire à une certaine forme de corrélation d'alertes ; elle repose sur des calculs probabilistes et ne constitue pas un vrai langage. Pour ADeLe, la corrélation entre alertes est définie explicitement par des contraintes temporelles et contextuelles. Pour LAMBDA, la corrélation est déduite automatiquement en cherchant une chaîne d'attaques élémentaires dont les post-conditions de l'une satisfont les préconditions de la suivante. Pour JIGSAW, la corrélation se fait comme pour LAMBDA mais se base sur une version abstraite des pré/post-conditions. Pour CRS, la corrélation est donnée explicitement à l'aide de contraintes temporelles et contextuelles entre les alertes.

1.2.5 Les langages de description d'alertes

Les langages de description d'alertes sont utilisés pour spécifier le format des alertes remontées par les IDS. Une alerte contient toutes les données utiles sur l'attaque qui a été détectée : la source de l'attaque, la cible, le type d'attaque, éventuellement une référence aux événements qui ont permis la détection, etc...A notre connaissance, CISL[8] [FKP+99] et IDMEF[9] [CD03] sont les seules références disponibles sur des langages de description d'alertes. CISL est maintenant abandonné au profit de IDMEF qui est en voie de standardisation auprès de l'IETF, puisqu'il vient d'être soumis à acceptation en tant que RFC. Le format IDMEF s'appuie sur le langage XML et les différents champs de l'alerte sont définis par une DTD.

1.2.6 Les langages de réaction

Les langages de réaction sont utilisés pour exprimer les éventuelles contre-mesures à prendre en réaction à la détection d'une attaque. Ces contre-mesures ont pour but de minimiser les conséquences de l'attaque en stoppant l'attaque en cours, en empêchant qu'elle se reproduise ou encore en corrigeant ses effets sur le système. Les IDS actuels qui disposent de fonctions de réaction n'ont pas de véritables langages pour les exprimer, ils utilisent plutôt des librairies propriétaires qui sont liées à l'IDS.

En dehors de la publication sur ADeLe dans [MM01], nous n'avons trouvé qu'une seule autre référence traitant d'un véritable langage de réaction. Gombault et Diop proposent, dans [GD02], une taxonomie des actions envisagées en cas de détection d'une attaque et présentent la mise en œuvre associée.

[8] Common Intrusion Specification Language

[9] Intrusion Detection Message Exchange Format

1.2.7 Récapitulatif des langages

Dans le tableau récapitulatif 1.1, nous dressons une liste (non exhaustive) des langages de description d'attaques qui ont été proposés. Nous avons choisi de ne pas y référencer les langages à usage général (Java, C, C++). Nous indiquons pour chaque langage dans quelle(s) classe(s) il intervient.

On peut remarquer que certains des langages représentés dans le tableau couvrent plusieurs aspects à la fois. Ainsi, le langage que nous proposons, ADeLe, intervient dans cinq classes : langages d'événement, d'exploit, de détection, de corrélation d'alertes et de réaction.

	exploit	événements	détection	description d'alertes	corrélation d'alertes	réaction
CASL [Sec98]	x					
NASL [Der99]	x					
IDLE [LMPT98]	x					
BSM [Sun]		x				
Tcpdump [JLM00]		x				
Syslog [Sys]		x				
Bishop [Bis95]		x				
IDIOT [KS95]			x			
RUSSEL [Mou97]			x			
MuSigs [LWJ98]			x			
BRO [Pax98]			x			
Snort [Roe99]			x			
SNP-L [TZ00]			x			
Sutekh [PD00]			x			
STATL [EVK00]			x			
LogWeaver [**?**]			x			
IDML [LTL01]			x			
$G^{A}_{S}SA_{T}A$ [Mé98]			x			
CISL [FKP+99]				x		
IDMEF [CD03]				x		
JIGSAW [TL00]					x	
CRS [MD03]					x	
LAMBDA [CO00]	x		x		x	
ADeLe [MM01]	x	x	x		x	x

TAB. 1.1 – Exemples de langages de description d'attaques (liste non exhaustive)

Il existe un certain nombre de langages de détection qui sont proches de nos travaux et qui permettent de corréler plusieurs événements ou alertes (figure 1.2). On peut les classer suivant plusieurs critères :

- selon que la signature est indépendante de l'algorithme de détection ou bien qu'elle est liée à l'algorithme (c'est-à-dire selon qu'on exprime ce que l'on cherche ou bien comment on le cherche) ,

- selon qu'ils permettent de corréler des événements, des alertes ou bien les deux (même si ce point est souvent plus lié à l'implémentation qu'à une impossibilité au niveau du langage).

On note ici que le langage ADeLe ainsi que le langage des Chroniques (utilisé par CRS) sont indépendants de l'algorithme et qu'il permettent de corréler à la fois des événements et des alertes. Même si les deux langages ont des pouvoirs d'expression similaires, ADeLe est dédié à la détection d'intrusions. Il nous semble que sa syntaxe le rend plus utilisable par un administrateur de sécurité parce que plus lisible et plus concise.

On notera par ailleurs, qu'en dehors de ces critères, LAMBDA et ADeLe sont des langages de description d'attaques complets (c-a-d qu'ils permettent d'exprimer l'attaque du point de vue de l'attaquant et du point de vue du défenseur). D'autre part, en LAMBDA, l'enchaînement temporel des alertes peut être déduit automatiquement des propriétés des pré/post-conditions.

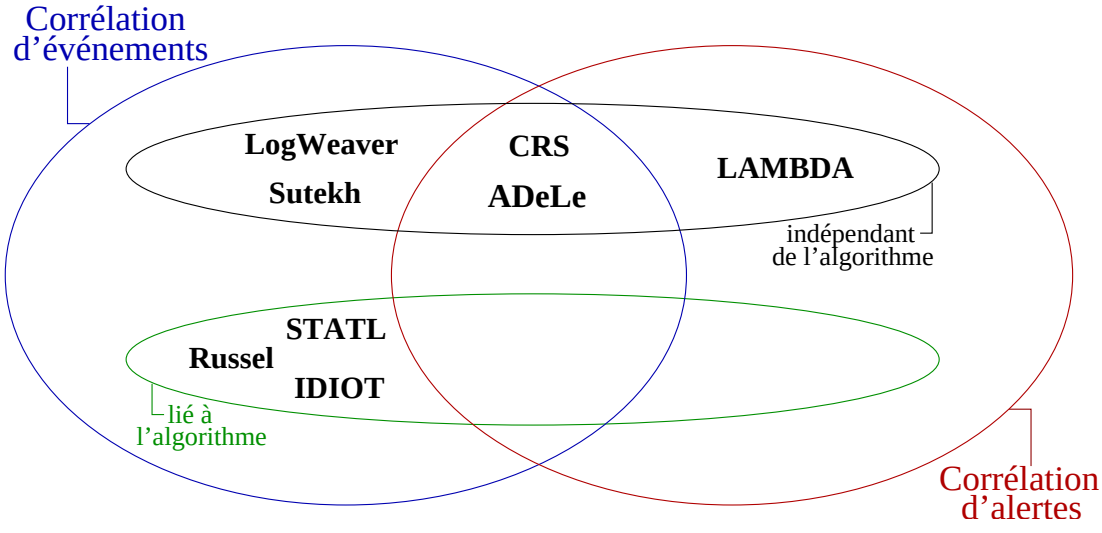

FIG. 1.2 – Langages de détection par corrélation d'événements/alertes

1.3 Position d'un IDS configurable en ADeLe dans la classification

Une partie du langage ADeLe est dédiée à la détection de l'attaque. Par la conception même du langage, on peut tirer quelques conclusions sur la position dans la classification d'un IDS qui serait configurable en ADeLe.

Sources de données : lors de la définition d'une signature, le langage ADeLe permet de désigner de façon homogène les données provenant de toutes les sources (système, réseau, application et alertes). Il n'y a donc pas de restriction sur les sources de données utilisées par un IDS configurable en ADeLe, si ce n'est qu'il faut déployer les capteurs ou sondes appropriés au sein du réseau à surveiller.

Méthode de détection : le langage ADeLe permet de donner une description complète pour chaque attaque connue. Il est donc clairement conçu pour des IDS basés sur l'approche par scénarios. D'ailleurs, si l'on considère un ensemble d'attaques que l'on aura décrites en ADeLe, il consitue (entre autres informations) une base de signatures.

Localisation de l'analyse : l'utilisation du langage ADeLe tend à privilégier des IDS qui effectuent une analyse centralisée puisque nous souhaitons corréler plusieurs événements (ou alertes) pouvant provenir de capteurs (ou sondes) différents. Toutefois, il y a des cas, au sein d'un système ou d'un réseau local, où la signature peut être vérifiée localement (même si les alertes résultantes seront centralisées afin d'avoir une vue globale sur l'ensemble du réseau). Effectuer l'analyse le plus tôt possible, c'est-à-dire lorsque l'on a accès à toutes les informations utiles pour réaliser le diagnostic, permet de réduire efficacement le flot des données au sein du système de détection d'intrusions.

Fréquence de l'analyse : dans une signature ADeLe, on ne présume pas de la fréquence d'analyse qui sera appliquée par l'analyseur (même si l'on peut exprimer des contraintes temporelles dans les signatures). ADeLe a été conçu pour décrire ce qui constitue l'attaque en faisant le moins d'hypothèses possible sur la façon de la détecter. Les opérateurs temporels des signatures ADeLe s'appliquent aussi bien dans le cas de l'analyse *offline* de fichiers de logs que dans le cas d'une analyse en continu des événements (ou alertes).

Comportement après détection : le langage ADeLe permet, pour chaque attaque décrite, d'exprimer une éventuelle réaction automatique en cas de détection. Un IDS configurable en ADeLe émet systématiquement une alerte et notifie l'opérateur. Par contre, il n'est pas obligé de réagir activement à la détection. Le

risque de provoquer un déni de service sur son propre réseau (en cas de fausse alerte suivie d'une réaction automatique) est relativement élevé. Il est plutôt souhaitable de proposer une réaction à un opérateur humain (qui prend la décision finale).

Chapitre 2

Syntaxe et sémantique informelle du langage ADeLe

Dans ce chapitre, nous présentons le langage ADeLe qui permet de décrire les différents aspects qui définissent une attaque : son mode opératoire, sa détection et une éventuelle réaction à l'attaque.

Nous présentons d'abord la structure globale d'une description ADeLe dans la section 2.1. Nous décrivons la partie `<EXPLOIT>` de la description qui rassemble les informations relatives à la mise en œuvre de l'attaque dans la section 2.2. Dans la section 2.3, nous détaillons la partie `<DETECTION>` de la description qui permet d'exprimer une signature pour détecter l'attaque. Dans la section 2.4, nous présentons la partie `<RESPONSE>` de la description qui permet d'exprimer une éventuelle réaction à l'attaque.

Pour chacune de ces parties, nous donnons la grammaire en notation EBNF[1] et la sémantique informelle associée. La plupart des exemples utilisés sont issus de la description d'une attaque nommée "NFS_mount" (cf annexe G) qui permet d'illustrer un grand nombre des concepts que nous allons présenter.

On notera que la sémantique opérationnelle de la partie détection est traitée séparément et fait l'objet du chapitre 3.

Pour donner une grammaire formelle de notre langage, nous nous inspirons de la notation EBNF et nous utilisons les conventions de notation suivantes :

- une règle de grammaire commence par un symbole non-terminal, suivi du symbole ':', de l'expression régulière correspondante et se termine par le symbole ';' ;
- tout symbole non-terminal est écrit en minuscules et représenté en italique ;
- tout symbole terminal est soit en majuscules, soit entouré par des doubles quotes (symbole '"', éventuellement précédé par le symbole '\' lorsqu'il apparaît lui-même dans le terminal) ;

[1] Extended Backus-Naur Form

– les parenthèses permettent de délimiter la portée des opérateurs '|', ' ?', '*' et '+' ;
– le symbole '|' représente l'alternative ;
– le symbole ' ?' désigne une expression facultative (0 ou 1 occurence) ;
– le symbole '*' désigne une répétition éventuellement nulle (0, 1 ou plusieurs occurences) ;
– le symbole '+' désigne une répétition (1 ou plusieurs occurences).

2.1 La description globale <ADELE>

L'agencement global d'une description d'attaque en ADeLe est présenté de façon synthétique dans la figure 2.1 (p. 31). Nous y avons indiqué de façon informelle à la fois la structure d'une description et la nature des informations contenues dans chacune de ses parties.

On peut noter que le squelette de la description (c'est-à-dire le contenant) forme un document XML valide[2] (dont la DTD se trouve à l'annexe B). Nous avons fait ce choix pour plusieurs raisons. Premièrement, une segmentation hiérarchique améliore la lisibilité. Ensuite, l'utilisation de balises XML simplifie l'extraction des différentes parties. Et finalement, utiliser un format texte facilite l'édition de la description en dehors de toute interface propriétaire.

La structure globale (c'est-à-dire au plus haut niveau) d'une description en ADeLe est définie par la grammaire suivante :

description : "<?xml version=\"1.0\"?>"
"<!DOCTYPE ADELE SYSTEM \"specadele.dtd\">"
"<ADELE"
"name=" STRING
("params=\"" (*list_param*)? "\"")? ">"
exploit
detection
response
"</ADELE>" ;
list_param : *param* ("," *param*)* ;
param : ("IN" | "OUT") TYPE (ARRAY)? VARNAME ;

La balise <ADELE> encapsule toute la description. Elle possède deux attributs.

[2]Pour des raisons de lisibilité, certains exemples de ce mémoire ne sont pas bien formés au sens XML car certains caractères (ex : '<') ne sont pas encodés.

```
<?xml version="1.0"?>
<!DOCTYPE ADELE SYSTEM "specadele.dtd">
<ADELE name="NomAttaque"
       params="IN Type1 Variable1, OUT Type2 Variable2 ...">

  <EXPLOIT>  #connaissance sur l'attaque du point de vue de l'attaquant
    <PRECOND>
      ... #conditions nécessaires pour lancer l'attaque
    </PRECOND>
    <ATTACK>
      <TEXT> ... </TEXT> #description des étapes de l'attaque
      <CODE> ... </CODE> #code source de l'attaque
    </ATTACK>
    <POSTCOND>
      ... #ce qu'a obtenu l'attaquant
    </POSTCOND>
  </EXPLOIT>

  <DETECTION> #connaissance sur l'attaque du point de vue du défenseur
    <DETECT>
      <EVENTS>
        ... #type d'événements/alertes observés durant l'attaque
      </EVENTS>
      <ENCHAIN>
        ... #ordre d'apparition des événements/alertes
            #et contraintes temporelles
      </ENCHAIN>
      <CONTEXT>
        ... #contraintes filtrant les événements/alertes intéressants
        ... #contraintes liant les événements/alertes d'une même attaque
      </CONTEXT>
    </DETECT>
    <CONFIRM>
      ... #vérification active du succès/échec de l'attaque
    </CONFIRM>
    <REPORT>
      ... #construction de l'alerte qui sera remontée (format IDMEF)
    </REPORT>
  </DETECTION>

  <RESPONSE> #connaissance sur la réponse à apporter à l'attaque
    ... #expression de la réaction envisagée après détection
  </RESPONSE>

</ADELE>
```

FIG. 2.1 – Vision synthétique d'une description d'attaque écrite en ADeLe

Le premier attribut, nommé "name", est obligatoire. Il constitue le nom courant associé à l'attaque qui sera utilisé dans l'alerte remontée en cas de détection de cette attaque. Le symbole terminal STRING correspond à une chaîne de caractères entourée de doubles quotes.

Le second attribut, nommé "params", est optionnel. Il peut contenir une suite de paramètres typés qui représentent des variables d'entrée ("IN") ou de sortie ("OUT"). Ces paramètres sont utilisés pour rendre la partie <EXPLOIT> réutilisable et participent à l'objectif de modularité d'ADeLe.

Si nous décrivons une attaque composée de plusieurs attaques mineures déjà décrites en ADeLe, il suffit ultérieurement de référencer (dans la sous-section <CODE> de <ATTACK>) le nom de ces attaques avec les paramètres appropriés pour invoquer leur code. Les paramètres d'entrée sont utilisés comme paramètres de l'attaque (ex : adresse IP cible). Les paramètres de sortie permettent de conserver les résultats de l'attaque (ex : login utilisateur compromis).

Le symbole terminal TYPE représente les types autorisés pour ces variables : IPAddr, String, Integer, Boolean, Connection. Des tableaux d'éléments de ces types peuvent également être utilisés (le symbole terminal ARRAY est facultatif et correspond à un nombre entre crochets). Le symbole terminal VARNAME correspond au nom de la variable que l'on souhaite déclarer.

Exemple provenant de l'attaque "NFS_Mount" :

```
<?xml version="1.0"?>
<!DOCTYPE ADELE SYSTEM "specadele.dtd">
<ADELE name="NFS_Mount" params="IN IPaddr targetip, OUT String
account, OUT Connection cnx">
...
</ADELE>
```

Dans le cas présent, on pourrait réutiliser cette description en invoquant NFS_Mount(192.168.1.2, login, handler) pour attaquer une hypothétique machine d'adresse IP "192.168.1.2" et recupérer le nom du login compromis ainsi qu'une connexion *rlogin* ouverte sous ce nom (via les variables login et handler).

Le corps de la description (compris entre les balises <ADELE> et </ADELE>) est composé de trois parties . Nous présentons d'abord la partie <EXPLOIT> en 2.2. Ensuite, nous détaillons la partie <DETECTION> en 2.3. Et finalement, nous décrivons la partie <RESPONSE> en 2.4.

2.2 La partie <EXPLOIT>

La première partie de la description d'attaque représente le point de vue de l'attaquant. Elle lie entre elles trois parties de l'attaque : les pré-requis nécessaires pour la lancer, son code (et/ou la description des étapes) et les résultats obtenus par l'attaquant. C'est pourquoi la partie <EXPLOIT> est décomposée en trois sections (qui contiennent respectivement les pré-conditions, le code et les post-conditions). La grammaire associée est :

exploit : "<EXPLOIT>"
precondition
attaque
postcondition
"</EXPLOIT>" ;

2.2.1 La section <PRECOND>

C'est dans cette section que l'on énumère les différents pré-requis pour que l'attaque réussisse. Il est important de pouvoir exprimer précisément les conditions qui font qu'une configuration est vulnérable[3]. En effet, comme les systèmes doivent être continuellement patchés avec des correctifs fournis par les éditeurs de logiciels et de systèmes d'exploitation, il est difficile de recréer dans un réseau de tests une configuration effectivement vulnérable (pour tester une attaque face à des IDS) si l'on ne dispose pas d'informations suffisamment détaillées.

Historiquement, cette section était présente pour représenter les niveaux de privilèges (issus de [Ken99]) nécessaires pour mener à bien l'attaque. Ensuite, elle a été enrichie par d'autres informations, dont certaines sont inspirées du langage LAMBDA [CO00].

Pour coder ces informations, nous proposons d'utiliser une liste de *mot-clé == "valeur"*. Si plusieurs valeurs sont possibles pour un des mots-clés, on marque l'alternative en les séparant avec le symbole '|'. Un mot-clé peut apparaître plusieurs fois si l'on veut marquer la conjonction (par exemple, deux logiciels qui doivent être installés en même temps).

Nous définissons ici plus formellement la syntaxe de cette section :

precondition : "<PRECOND>"
(KEYWORD "==" STRING ("|" STRING)*)*
"</PRECOND>" ;

[3]On peut noter que certaines attaques ne reposent pas sur l'activation d'une vulnérabilité particulière (ex : sabotage par un administrateur système possédant tous les droits d'accès).

Nous donnons ici une liste non exhaustive des mots-clés (symbole terminal KEYWORD) que nous proposons pour coder ces informations[4] :

- **TargetOS** : dénomination usuelle du ou des systèmes d'exploitations affectés. On utilise la convention suivante pour préciser les numéros de version concernés : ils sont indiqués entre parenthèses et séparés par des virgules ('_' signifie version originale, '*' signifie toutes les versions).
- **InstalledSoftware** : nom et numéro de version du logiciel (ou de la librairie dynamique) qui doit être installé.
- **ExpertiseLevel** : niveau d'expertise requis pour mettre en oeuvre l'attaque (respectivement `Novice`, `Confirmed` et `Expert`)
- **AccessLevel** : niveau de privilège requis pour exécuter l'attaque. Nous utilisons les niveaux proposés dans [Ken99] (`Remote`, `Local`, `User`, `Root` et `Physical`). Ils désignent respectivement : un accès distant à la cible par le réseau, un accès au réseau local de la cible, un accès à la cible en tant que simple utilisateur, un accès à la cible en tant qu'administrateur et un accès physique à la machine cible.
- **VulnerabilityID** : identifiant correspondant à une référence standard à la vulnérabilité (dans les bases de données CERT/CC, BugTraq, CVE, X-Force).

Ainsi, pour représenter une hypothétique attaque contre Internet Explorer 5.5 (sous plusieurs versions de Microsoft Windows), on pourrait écrire :

```
<PRECOND>
 TargetOS          == "Windows 98 (_,SE)"
                    | "Windows ME (_)"
                    | "Windows NT Workstation 4.0 (*)"
 InstalledSoftware == "Microsoft Internet Explorer 5.5 (SP2)"
 ExpertiseLevel    == "Confirmed"
 AccessLevel       == "Remote"
 VulnerabilityID   == "Cert VU#932283"
                    | "Bugtraq BID-4080"
                    | "Cve CAN-2002-0022"
                    | "Xforce ie-html-directive-bo(8116)"
 ...
</PRECOND>
```

On peut noter que la plupart de ces catégories ont un but plus informationnel qu'opérationnel. Notamment, nous n'avons pas poussé le détail jusqu'au niveau atteint dans le langage LAMBDA où chaque information utile à l'attaque (pré-condition et post-condition) est codifiée sous une forme proche des faits Prolog.

[4] voir la base de vulnérabilités ICAT (http://icat.nist.gov) pour d'autres exemples de champs

L'intérêt d'une telle approche est d'inférer un enchaînement (potentiellement inconnu) d'attaques élémentaires uniquement en les mettant en relation deux à deux de telle façon que les post-conditions de l'une satisfont les pré-conditions de l'autre.

2.2.2 La section <ATTACK>

C'est dans cette section que nous allons pouvoir inclure le code source de l'attaque (écrit dans n'importe quel langage). Cela permet de disposer d'un moyen de rejouer l'attaque face à des IDS pour tester leur efficacité de détection.

attaque : "<ATTACK>"
(*text*)?
(*code*)*
"</ATTACK>" ;
text : "<TEXT>" (TEXT)? "</TEXT>" ;
code : "<CODE" "language=" STRING
("filename=" STRING)? ">"
(TEXT)?
"</CODE>" ;

Une première balise facultative, nommée <TEXT>, permet d'annoter la description de l'exploit. Et dans l'éventualité où l'on ne dispose pas du code de l'attaque, on peut y mettre une description textuelle du mode opératoire de l'attaque. Le symbole terminal TEXT représente n'importe quel texte.
Exemple :

```
<TEXT> L'attaquant doit d'abord se connecter à ...</TEXT>
```

Une seconde balise, nommée <CODE>, est chargée de contenir le code source de l'attaque. Son attribut *language* permet de définir le type de langage utilisé afin qu'on puisse exécuter le code de l'attaque (grâce à un compilateur ou un interpréteur adapté au langage utilisé). Cela peut être un langage de programmation à usage général (tel que **C**, **C++**, **Perl**, etc) ou bien un langage dédié à l'écriture d'attaques (tel que **Casl** [Sec98] ou bien **NASL** [Der99]).

Son attribut *filename* contient le nom du fichier initial (qui sera également le nom utilisé pour extraire le code vers un fichier). On notera que la balise <CODE> peut être répétée de zéro à plusieurs fois (pas de code ou code source réparti en plusieurs fichiers).
Exemple :

```
<CODE language="C" filename="exploit.c"> ... </CODE>
```

Nous avons défini un langage de script haut niveau dédié à l'écriture du code des attaques : EDL, acronyme de *Exploit Description Language*. Ce langage ne dispose pas, à l'heure actuelle, de compilateur ou d'interpréteur. Pour autant, n'importe quel autre langage peut être utilisé pour écrire un exploit.

Nous ne donnerons pas ici une grammaire détaillée de ce langage car il repose sur les concepts classiques que l'on trouve dans la plupart des langages de programmation impératifs :

- données typées (`String`, `Integer`, `Boolean`, `IPaddr` ...) et tableaux,
- déclaration et assignation (' `:=`') de variables,
- expressions booléennes (`&&`, `||` ...),
- opérateurs mathématiques (`+`, `-`, `*`, `/`, `%` ...),
- fonctions de manipulation de chaînes de caractères (`strlen()`, `strcat()` ...),
- opérateur de *pattern matching* (`MATCHES`) utilisant les expressions régulières,
- exécution conditionnelle des actions[5] :
  `IF (`*expression_booleenne*`) {`(*action*)*`}`
  `ELSE {`(*action*)*`}`
- exécution itérative des actions :
  `WHILE (`*expression_booleenne*`) DO {`(*action*)*`}`
- appels de fonctions (avec paramètres et valeur de retour).

Ce langage met l'accent sur l'utilisation d'une librairie de fonctions (système et réseau) prédéfinies. Ces fonctions peuvent être, selon le type d'attaque à coder, de bas niveau ou de haut niveau. Pour décrire l'enchaînement des actions qui constituent l'attaque, nous utilisons cette librairie de fonctions prédéfinies.

Par exemple, dans la catégorie des fonctions système,

```
return_value = Exec_shell_cmd(shell_command,output);
```

permet d'exécuter une commande quelconque dans le shell, d'obtenir le résultat de sa sortie standard et de savoir si elle s'est exécutée sans erreurs.

Pour coder des attaques réseau, on peut, par exemple, aller de l'envoi bas-niveau d'un datagramme IP particulier :

```
return_value = Network_send_packet(ip_packet);
```

jusqu'à l'établissement d'une connexion FTP et l'utilisation des commandes haut niveau du protocole (telles que USER, PASS ...) :

[5] *action* peut désigner une commande élémentaire (assignation, appel de fonction) mais aussi une conditionnelle (IF) ou une itération (WHILE).

```
return_value = Ftp_connect(target_ip,ftp_port,cnx_handler,output);
IF (cnx_handler != NULL){
  retval1 = Ftp_cmd(cnx_handler,"USER "+target_login,output);
  retval2 = Ftp_cmd(cnx_handler,"PASS "+target_passwd,output);
}
```

Pour mettre en oeuvre une telle librairie de fonctions, il est possible de s'appuyer sur des API existantes :

- pour la partie purement réseau des attaques (protocoles de communication, création de paquets, interception de paquets ...), on peut utiliser, entre autres, **Lcrezo**[6], **Libnet**[7] et **Libpcap**[8] ;
- pour l'écriture d'exploits, il existe depuis peu une API qui facilite le codage des attaques de type *Buffer Overflow* et *Format String* : **LibExploit**[9] ;
- on peut également tirer parti de la mise en oeuvre existante du langage **NASL**[10] (utilisé par le scanner de vulnérabilités Nessus) qui est conçu pour le test de vulnérabilités.

L'objectif du langage EDL est de ne pas se limiter à l'aspect réseau (comme le fait NASL) car on souhaite pouvoir écrire des attaques à la fois systèmes et réseaux. Les exploits utilisés dans les exemples de descriptions d'attaque de ce mémoire (annexe G) sont écrits dans ce langage.

Opérateurs d'action

Nous savons que l'écriture d'une signature d'attaque pour la détection est rendue difficile lorsqu'il existe plusieurs variantes de cette attaque. Comme nous nous plaçons du point de vue de l'attaquant dans cette section, il est intéressant de pouvoir factoriser le code de plusieurs attaques similaires (qui donneront lieu à la même alerte). Cela permet notamment de réduire la taille de la base d'attaques permettant de tester l'efficacité des IDS.

Ainsi, pour obtenir des scripts d'attaque génériques (ie comportant des variantes), nous proposons l'utilisation de méta-opérateurs (annotant le code) qui vont influer sur l'exécution. Celle-ci se fait toujours séquentiellement, c'est-à-dire dans l'ordre d'écriture des commandes au sein du code de l'attaque, sauf lorsqu'il y a application de l'un de ces deux opérateurs : `Non_ordered` et `One_among`.

[6] http://www.laurentconstantin.com/fr/lcrzo/
[7] http://www.packetfactory.net/Projects/Libnet/
[8] http://www.tcpdump.org
[9] http://www.packetfactory.net/projects/libexploit/
[10] http://www.nessus.org

Non_ordered

Dans certaines attaques, il est possible d'intervertir l'ordre de plusieurs étapes sans changer la validité de l'attaque. Ainsi, l'opérateur `Non_ordered` permet d'exprimer que plusieurs sections de code (encadrées par des crochets) doivent toutes être exécutées mais dans un ordre quelconque. Exemple :

```
Non_ordered{
 [ action1; ]
 [ action2; ]
 [ action3; ]
}
```

One_among

Si, pour l'une des étapes composant une attaque, on a le choix entre plusieurs actions (ou groupes d'actions) équivalentes en termes de conséquence, alors l'utilisation de l'opérateur `One_among` permet de n'écrire qu'une seule attaque (qui représente plusieurs variantes). Cet opérateur ne permet l'exécution que d'une seule des alternatives (représentée par un bloc de code entre crochets). Exemple :

```
One_among{
 [ action1; ]
 [ action2; ]
}
```

Lorsqu'une attaque (écrite en EDL) contient l'un des opérateurs d'action, il faudra faire un choix à l'exécution. Différentes stratégies peuvent être envisagées pour choisir la variante qui sera utilisée dans l'attaque : ordre aléatoire, ordre inverse, toujours le premier choix, ...

Un lien entre attaque et détection

La partie `<DETECT>` d'une description ADeLe (cf 2.3.1, p.42) contient les informations permettant de détecter l'attaque à partir des activités surveillées au sein du système. Comme les événements (issus des capteurs) et les alertes (issues des sondes) correspondent à des étapes observables de l'attaque, il est intéressant d'établir un lien entre les actions et leur détection.

Si l'on dispose de l'information nécessaire, il est possible de mettre en relation une action (ou un groupe d'actions) observable avec les événements/alertes qu'elle doit générer. Plus concrètement, cela consiste à délimiter la portion de code correspondant à l'événement/alerte et de lui donner le même nom que celui qui sera utilisé plus loin dans la signature (partie `<DETECT>`).

Par exemple, si l'on peut observer sur le réseau (grâce à un capteur ou à une sonde) une requête de type `finger` vers une machine cible, on annote le code en utilisant le mot-clé `EVENT` suivi du nom qui sera utilisé par la suite (ici `E2`) pour

désigner cette instance d'événement/alerte :

```
EVENT E2{
  Exec_shell_cmd("finger @"+iptarget,users_list,ret_val2);
}
```

Dans certains cas, une action réalisée par l'attaquant n'est pas observable car elle n'entre pas dans le champ de la détection (ex : elle ne laisse aucune trace dans le réseau local surveillé). Par conséquent, elle ne pourra pas être associée à un événement ou une alerte.

2.2.3 La section <POSTCOND>

La dernière section de <EXPLOIT> contient les informations relatives aux conséquences de l'attaque si elle réussit.

Pour coder ces informations, nous proposons d'utiliser une liste de *mot-clé* := *"valeur"*. Un mot-clé peut apparaître plusieurs fois si l'on veut marquer la conjonction (par exemple, une attaque qui donne lieu simultanément à plusieurs types de conséquences).

La syntaxe est donnée par la grammaire suivante :

postcondition : "<POSTCOND>"
(KEYWORD ":=" STRING)*
"</POSTCOND>" ;

Pour préciser le type de conséquences de l'attaque, nous préconisons l'utilisation du mot-clé **UnauthorizedResult**. Il est inspiré de la taxonomie des incidents de sécurité présentée dans [HL98] qui propose cinq types généraux pour classer les conséquences d'une attaque. Ils correspondent aux différents types de résultats obtenus illégalement par l'attaquant.

Les cinq types proposés pour les conséquences sont :

- "Increased access" : élévation de privilèges non autorisée au niveau de l'accès à une machine ou un réseau ;
- "Disclosure of information" : diffusion d'information à toute personne n'étant pas autorisée à accéder à cette information ;
- "Corruption of information" : modification non autorisée de données sur une machine ou dans un réseau ;
- "Denial of service" : dégradation ou blocage intentionnel des ressources liées à un ordinateur ou un réseau ;
- "Theft of resources" : utilisation non autorisée des ressources liées à un ordinateur ou un réseau ;

Nous utilisons le mot-clé **AccessLevel** (comme dans la section <PRECOND>) pour désigner le niveau d'accès à la cible [Ken99] obtenu par l'attaquant.

On peut également ajouter des mots-clés pour préciser la conséquence contenue dans **UnauthorizedResult**. Par exemple, dans le cas d'un vol de ressources, on peut utiliser le mot clé **Resource** pour nommer la ressource concernée ("Cpu", "Network bandwith", "Storage" ...).

Dans le cas de l'attaque NFS_Mount (annexe G.1), on a simultanément une augmentation des privilèges de l'attaquant (compte utilisateur compromis) et une altération de l'information (fichier ".rhosts" modifié).

Cela se traduit par :

```
<POSTCOND>
  AccessLevel        := "USER" # niveau d'acces obtenu
  UnauthorizedResult := "Increased access"
  UnauthorizedResult := "Corruption of information"
</POSTCOND>
```

2.3 La partie <DETECTION>

Après avoir montré, dans la partie <EXPLOIT>, comment on peut représenter l'attaque du point de vue de l'attaquant, nous nous plaçons maintenant du point de vue du défenseur. La seconde partie d'une description en ADeLe (<DETECTION>) contient les informations relatives à la détection.

N.B. : Sauf indication contraire et jusqu'à la fin de ce chapitre, chaque occurence du mot "événement" désigne indifférement les messages issus des capteurs («événements») que les messages issus des sondes (alertes). Cela tient au fait que notre langage utilise une notation homogène pour accéder aux valeurs contenues dans ces messages (cf 2.3.1.1).

Nous avons été amenés à faire un certain nombre de choix concernant l'architecture logicielle nécessaire à l'utilisation de la partie détection de notre langage.

Disponibilité et format des événements

La liste des événements observables, et donc la liste des événements utilisables dans nos signatures, dépend des capteurs et des sondes qui sont déployés dans les systèmes et réseaux surveillés. Cette liste constitue un dictionnaire d'événements.

Les événements remontés par des sondes (configurables ou non en ADeLe)

doivent être au format IDMEF[11][CD03] développé par l'IDWG qui se base sur le langage XML.

Pour les événements remontés par des capteurs, nous proposons le format EVMEF qui s'appuie sur le langage XML (cf annexe E). Il est proche de l'IDMEF mais adapté à la description d'événements issus du réseau, des systèmes ou des applications. Ce choix permet de rendre homogène l'accès aux valeurs intrinsèques des événements (qu'ils proviennent de capteurs ou bien de sondes). Lorsqu'il n'est pas possible d'utiliser ce format en sortie des capteurs, il faut mettre en place une couche d'adaptation. Elle traduit, à l'exécution, la notation EVMEF pour accéder aux valeurs contenues dans les événements en son équivalent adapté au format de sortie. Cela permet de continuer à utiliser la notation EVMEF dans les signatures.

Ordonnancement correct des événements

Le processus de détection induit par le langage ADeLe est basé sur des contraintes temporelles entre événements. Nous devons donc assurer que leur ordre relatif d'apparition (dans le flux des événements entrants proposés à la détection) est correct.

Pour cela, diverses solutions complémentaires peuvent être envisagées. Tout d'abord, l'horodatage des événements doit avoir la même référence : il faut donc mettre en place une stratégie de correction de l'heure locale des équipements (ex : utilisation d'un serveur de temps[12]). Ensuite, la réception des événements par un analyseur (même correctement datés et expédiés sans délai) risque de se faire dans un ordre différent de leur émission (délais d'acheminement différents selon le chemin réseau, ...). On peut utiliser une file d'attente intermédiaire pour réordonner les événements dans un ordre temporel croissant.

L'ensemble des informations ayant trait à la détection est réparti dans trois sections distinctes. Cette séparation se traduit au niveau de la syntaxe par la grammaire suivante :

detection : `"<DETECTION>"`
detect
confirm
report
`"</DETECTION>"` ;

La section `<DETECT>` (2.3.1) contient la signature qui permettra de détecter l'attaque en termes de corrélation d'événements/alertes. La section `<CONFIRM>`

[11] *Intrusion Detection Message Exchange Format* : en cours de standardisation car candidat au statut de RFC

[12] de type NTP (*Network Time Protocol*)

(2.3.2) permet d'exprimer un test sur la réussite ou l'échec de l'attaque afin d'affiner le diagnostic de détection. La section `<REPORT>` (2.3.3) contient les informations nécessaires à la construction de l'alerte au format IDMEF qui sera émise en cas de détection de l'attaque.

Nous détaillons maintenant la syntaxe associée à chacune de ces sections.

2.3.1 La section `<DETECT>`

Dans cette section de la description ADeLe, nous proposons un langage de haut-niveau qui permet d'exprimer la détection d'une attaque en donnant explicitement une signature de cette attaque basée sur la corrélation entre plusieurs événements. Nous entendons par le terme «corrélation» la mise en relation de plusieurs événements qui ont été générés durant la même attaque, sur des critères temporels et contextuels. Ce processus de détection par corrélation intervient **après** les éventuelles phases d'aggrégation[13] et de fusion[14] d'événements.

Nous verrons dans le chapitre 3 (dédié à la sémantique opérationnelle de la détection) que l'on peut compiler cette signature pour obtenir un automate de reconnaissance.

Nous avons choisi de scinder en trois parties l'écriture de la signature. La séparation des informations qui sont de nature différente permet à la fois de simplifier la grammaire et de rendre la description plus lisible. La grammaire reflète la déclaration en trois temps de la signature :

detection : "<DETECT>"
events
enchain
context
"</DETECT>" ;

La sous-section `<EVENTS>` (2.3.1.1) contient la définition et le nommage des événements intéressants qui doivent être observés lorsque l'attaque se produit. La sous-section `<ENCHAIN>` (2.3.1.2) permet de donner les contraintes sur l'ordre d'apparition de ces événements (à l'aide d'opérateurs de séquence, d'alternative, de conjonction et d'absence d'événements) ainsi que sur les intervalles temporels qui les séparent. La sous-section `<CONTEXT>` (2.3.1.3) permet la définition d'un ensemble de contraintes sur les valeurs contenues dans les événements (pour filtrer certains événements ou relier ceux qui appartiennent à la même attaque).

[13] regroupement en cluster d'événements ayant des caractéristiques similaires (ex : même adresse IP source).

[14] fusion de plusieurs événements distincts (correspondant à l'observation par plusieurs capteurs/sondes redondants de la même activité suspecte) en un seul.

2.3.1.1 La sous-section <EVENTS>

Dans cette première partie de la signature, nous définissons un ou plusieurs types d'événements qui seront utilisés dans le reste de la signature (dans les sous-sections <ENCHAIN> et <CONTEXT>).

Cela se traduit par la grammaire suivante :

events : "<EVENTS>"
(*event*)+
"</EVENTS>" ;

Le flux des événements entrants constitue la matière première de notre processus de détection. Comme chaque description ADeLe ne concerne qu'un seul type d'attaque, il faut filtrer ce flux et ne garder que les types d'événements qui sont susceptibles de la concerner. Par exemple, si une attaque se détecte au niveau système, on ne doit pas perdre du temps à considérer des événements provenant de sources de données réseau. C'est pourquoi on va définir un filtre pour chaque type d'événement intéressant et donner un nom à ce type. Nous précisons que les noms de types associés aux filtres sont locaux à chaque attaque.

Si l'on veut pouvoir distinguer deux événements et comparer les valeurs contenues dans les champs de ces événements pour les corréler, il faut d'abord pouvoir les nommer. On déclare donc des noms associés aux instances des événements en même temps que la définition du type.

Ainsi, chaque symbole non-terminal *event* est défini par :

event : ID ":" ID "{" (FIELD *operator* FIELD
| FIELD (*operator* | "MATCHES") STRING) ";")+
"}" ID ("," ID)* ";" ;

operator : ("==" | "!=" | ">" | "<" | ">=" | "<=") ;

La première occurence du symbole terminal ID correspond au nom du type d'événement que l'on est en train de définir (ex : "RPCINFO").

La seconde occurence de ID, derrière le caractère ':', précise le type de source de données concerné (ex : "PACKET" pour des paquets réseau, "BSM" pour de l'audit système Solaris, "SNARE" pour de l'audit système Linux, "IDMEF" pour les alertes provenant de sondes ...) et constitue un premier niveau de définition du filtrage. C'est ce même type qui détermine le format des données brutes contenues dans le type d'événement que nous définissons.

Les différentes lignes contenues entre "{" et "}" constituent le deuxième niveau de définition du filtrage. Ce sont les contraintes internes que doit vérifier chaque événement entrant pour être retenu et classifié selon le type courant. Nous avons défini deux sortes de contraintes :

- relation (symbole non-terminal *operator*) entre deux champs (symbole terminal `FIELD`) du même événement ;
- relation entre un champ de l'événement et une valeur (symbole terminal `STRING`).

Le symbole non-terminal *operator* représente une alternative entre plusieurs opérateurs infixes (resp. égalité, différence, supériorité stricte, infériorité stricte, supériorité et infériorité). L'égalité et la différence portent sur la valeur ASCII tandis que les autres relations portent sur les valeurs numériques (après transformation).

Le symbole terminal `"MATCHES"` représente une relation de test d'expressions régulières. Le symbole terminal `STRING` correspond à une chaîne de caractères quelconque entourée par des double-quotes.

Le symbole terminal `FIELD` désigne l'accès à la valeur textuelle d'un champ XML de l'événement (au format IDMEF ou EVMEF). Lorsque les capteurs n'utilisent pas le format EVMEF en sortie, un convertisseur doit être utilisé pour traduire l'accès au champ correspondant et en extraire la valeur. Les champs XML considérés sont soit des noeuds textuels, soit des attributs.

Nous utilisons la notation XPATH [Wor99] abrégée pour représenter le chemin d'accès à un champ XML particulier d'un événement. Nous avons ajouté deux règles qui dérogent à cette notation :

- le préfixe initial est omis (`"/IDMEF-Message/"` pour le format IDMEF et `"/Event-Message/"` pour le format EVMEF) ;
- pour l'accès à un noeud textuel, le suffixe final est omis (`"/text()"`)

Par exemple, on utilisera la notation `"Network/packet/ether/ip/tcp/dport"` pour obtenir la valeur du port destination d'un paquet réseau appartenant à une connexion TCP. De même, pour obtenir la chaîne qui identifie l'analyseur ayant émis une alerte, on accède au champ correspondant (attribut) avec la notation `"Alert/Analyzer@ident"`.

Pour les événements au format IDMEF, le champ `Alert/Classification[0]/-name` correspond toujours au nom associé à l'alerte remontée par une sonde. On peut donc utiliser ce premier critère pour les filtrer.

Pour les événements provenant de capteurs, il y a plusieurs cas de figure. Si un capteur utilise le format EVMEF en sortie et n'effectue aucun typage préalable, alors le nom associé à chacun des événements qu'il délivre est générique (ex : le champ `Network/Classification[0]/name` vaut "packet" pour tous les paquets) et ne permet pas de discriminer les événements. Si ce même capteur effectue un

typage préalable, alors on peut utiliser le nom associé pour filtrer nos événements (ex : `Network/Classification[0]/name == "Rpcinfo"`). Si le capteur utilise un autre format de sortie, alors on n'utilise pas le champ `Classification[0]/name` pour faire ce deuxième filtrage.

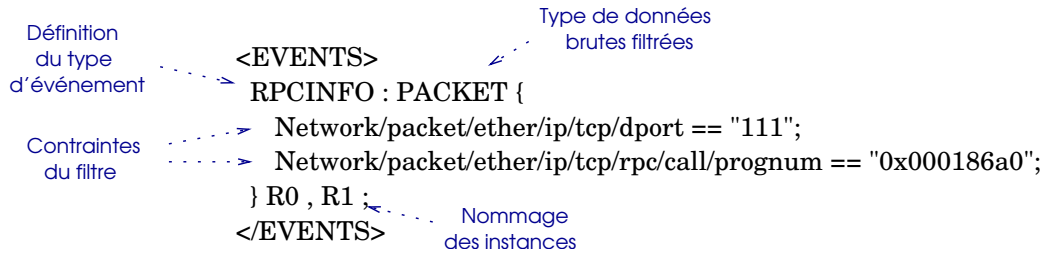

FIG. 2.2 – Exemple de définition et déclaration d'événements

La fin de la règle *event*, après le symbole `"}"`, permet de déclarer un ou plusieurs noms d'instances (symboles terminaux `ID`) d'événements correspondant au type que l'on vient de définir. Cela signifie que l'on va pouvoir, dans le reste de la signature, désigner un événement particulier grâce au nom qui lui est associé. Cela permet de donner des contraintes sur l'enchaînement temporel de plusieurs événements (sous-section `<ENCHAIN>`) mais aussi sur les valeurs contenues dans ces événements (sous-section `<CONTEXT>`).

Dans l'exemple de la figure 2.2, on peut voir la définition du type d'événement "RPCINFO" (local à l'attaque) qui constitue un filtre sur une source de données de type "PACKET" (paquets réseau) ainsi que la déclaration de deux instances d'événements nommées "R0" et "R1".

N.B. : si l'on définit plusieurs types d'événements différents utilisant la même source de donnée mais dont les filtrages ne portent pas sur les mêmes champs, alors il y a un risque que le même événement concret corresponde à plusieurs types à la fois. Afin que le typage d'un événement soit une opération déterministe, nous avons fait le choix arbitraire de ne considérer que le premier type qui convient (dans l'ordre des définitions). Toutefois, il est possible de déterminer ce type de situation dès la compilation et de mettre en garde l'utilisateur sur l'ambiguïté de sa description.

2.3.1.2 La sous-section `<ENCHAIN>`

Dans cette seconde partie de `<DETECT>`, nous définissons la signature temporelle qui permettra de détecter l'attaque à partir de plusieurs événements. Elle permet de donner toutes les informations utiles sur l'ordre dans lequel les événements doivent apparaître ou les délais qu'ils doivent respecter.

Avant de comparer des événements suivant un critère temporel, nous rappelons les conventions liées au temps dans le processus de détection (cf p. 41). Nous faisons l'hypothèse que le flux des événements entrants garantit que chaque événement y apparait avec un horodatage supérieur (ou égal) au précédent. Pour horodater un événement, réputé instantané, nous prenons pour référence la date contenue dans son champ XML `DetectTime`[15].

Nous utilisons le terme «hypothèse» pour désigner une reconnaissance partielle de la signature. Une nouvelle hypothèse vide est créée à chaque fois qu'apparaît l'un des premiers événements possibles de la signature. Chaque hypothèse contient l'historique des événements qui l'ont fait progresser dans la reconnaissance de cette signature. La détection de l'attaque est avérée lorsque la reconnaissance de la signature est complète (c'est-à-dire dès que l'hypothèse contient tous les événements attendus dans la signature). On doit alors émettre une alerte construite à partir des événements contenus dans l'hypothèse.

La signature temporelle, contenue dans la balise `<ENCHAIN>`, est donnée par la grammaire suivante :

enchain : `"<ENCHAIN>"`
scenario
(*timeout*) ?
(*mindelay* | *maxdelay*)*
`"</ENCHAIN>" ;`

Le premier type de contrainte temporelle décrit l'ordre partiel d'apparition des événements dans le flux des événements entrants. C'est une expression composée de noms associés à des instances d'événements[16] (précédemment définis dans la sous-section `<EVENTS>`) auxquels on a pu appliquer différents opérateurs. La grammaire associée à cette expression est :

scenario : *item*

[15] ce champ n'étant pas obligatoire dans le format IDMEF, il faudra le cas échéant utiliser le champ `Alert/CreateTime`.

[16] sauf dans le cas particulier d'une répétition où l'instance est anonyme et représentée par le nom associé au type.

```
          | suite ;

item : ID
       | oneamong
       | nonordered
       | without ;

suite : "(" ( item ( " ;" ( item | ID "^" NUMBER ) )+
            | ID "^" NUMBER ( " ;" ( item | ID "^" NUMBER ) )*
            )
        ")" ;
```

Le symbole non-terminal *scenario*[17] représente l'ordonnancement temporel entre les événements. Il correspond soit au symbole non-terminal *item*, soit au symbole non-terminal *suite*.

Un *item* représente soit l'apparition d'une instance d'événement nommée (symbole terminal `ID` préalablement défini dans la section `<EVENTS>`), soit la reconnaissance d'un sous-scénario représenté par un opérateur (*oneamong*, *nonordered* ou *without*).

Une *suite* représente un scénario équivalent soit à une répétition (`ID "^" NUMBER`), soit à une séquence (`" ;"`) de plusieurs sous-scénarios (répétition ou *item*).

La répétition de plusieurs événements du même type (`ID "^" NUMBER`) est une notation abrégée de la séquence où l'on ne considère que le type de l'événement sans nommer les instances. Le symbole terminal `ID` correspond à un type d'événement et le symbole terminal `NUMBER` indique le nombre de répétitions. Les instances d'événements ainsi définies sont anonymes et ne pourront faire l'objet d'aucune contrainte. Du point de vue de la sémantique, le scénario `(RPCINFO^3)` est équivalent à `(RPCINFO ; RPCINFO ; RPCINFO)`, c'est-à-dire qu'il représente une séquence de trois événements de type `RPCINFO`. Toutefois, cette seconde notation n'est pas autorisée dans la syntaxe car les événements (symbole terminal `ID`) qui apparaissent dans la règle *item* doivent être des noms d'instances. Lors de la construction d'un automate de reconnaissance, une répétition est automatiquement traduite en une simple séquence d'événements (dont les instances ne sont pas nommées).

Opérateur de séquence `" ;"`

La règle *suite* introduit l'opérateur de **séquence** (`" ;"`). Il permet de décrire

[17] N.B. : par la suite, nous utilisons le terme «scénario» pour désigner cette expression qui constitue un sous-ensemble de la signature complète.

les attaques composées de plusieurs étapes dont l'ordre d'apparition des événements est connu et fixé d'avance.
Par exemple, soient trois événements `A0`, `B0` et `C0` (éléments *item* réduits à un seul événement) dont la reconnaissance doit apparaître dans le flux des événements dans un ordre précis, on écrira :

```
<ENCHAIN> (A0 ; B0 ; C0) </ENCHAIN>
```

Concrètement, cela signifie que l'on cherche dans le flux des événements toutes les instances de triplets d'événements concrets dont l'ordre d'apparition correspond et où chaque élément vérifie les contraintes portées respectivement par `A0`, `B0` et `C0`. L'ordre d'apparition dans le flux des événements est **relatif** : chacun des éléments du triplet peut être séparé du suivant dans le flux d'événements par un nombre arbitraire d'événements intermédiaires. La séquence signifie "suivi de" et non pas "suivi immédiatement de".

On peut noter que la grammaire de *suite* n'autorise pas les séquences de séquences (c'est-à-dire des parenthèses imbriquées). En effet, l'opérateur de séquence tel que nous le définissons en 3.3.2 (p. 71) est associatif : $((a\,;\,b)\,;\,c) \equiv (a\,;\,(b\,;\,c)) \equiv (a\,;\,b\,;\,c)$. On peut donc remettre tous les éléments *item* au même niveau dans une seule expression parenthèsée sans rien changer à la sémantique de l'opérateur.

Opérateur de disjonction `One_among`

Une description ADeLe est dédiée à une attaque particulière. On doit donc écrire autant de descriptions que d'attaques différentes. Toutefois, il arrive qu'une attaque comporte plusieurs variantes mineures. Cela consiste généralement en la substitution de l'une des étapes de l'attaque par une autre (qui est équivalente en termes de conséquences mais qui génère des traces différentes).

L'opérateur `One_among`[18] permet, dans le scénario, de factoriser en une seule expression les variantes qui ont lieu à la même étape.

La syntaxe associée est :

oneamong : `"One_among" "{"` *scenario* `","` *scenario*)+ `"}" ;`

L'opérateur est représenté par le symbole terminal `"One_among"` qui encadre plusieurs expressions entre des accolades. On note que ces expressions représentent elles-mêmes des (sous-)scénarios puisque la règle *scenario* est appelée récursivement.

Exprimé de manière informelle, le scénario associé à un opérateur `One_among` est reconnu dès que l'un de ses sous-scénarios est reconnu (indépendamment

[18] équivalent (côté détection) de l'opérateur d'action `One_among` précédemment introduit.

des autres). Cet opérateur permet donc de n'avoir à écrire qu'une seule section `<ENCHAIN>`. La sémantique opérationnelle associée à cet opérateur (cf 3.3.3) permet de le considérer comme une simple étape du scénario global en ne traitant qu'une seule fois ce qui se trouve avant l'opérateur.

Par exemple, dans le scénario suivant :

```
<ENCHAIN> (A0 ; One_among{B0 , C0}) </ENCHAIN>
```

on souhaite reconnaître soit `"(A0 ; B0)"`, soit `"(A0 ; C0)"`. La reconnaissance de `A0` est commune aux deux alternatives, ce qui n'aurait pas été le cas si l'on devait, au moment de la détection, redévelopper le scénario en plusieurs `<ENCHAIN>` indépendants (ne contenant plus d'alternatives).

Opérateur de conjonction `Non_ordered`

Lorsqu'une attaque comporte plusieurs étapes dont l'ordre est interchangeable, on ne sait pas, au niveau de la détection, quel sera l'ordre précis d'apparition des événements générés lors de chacune des étapes. De plus, ces étapes peuvent être entrelacées dans le flux des événements. Par conséquent, si l'on utilise seulement l'opérateur de séquence, on est obligé de décrire tous les ordonnancements possibles, ce qui n'est pas envisageable dans la pratique.

L'opérateur `Non_ordered`[19] permet d'exprimer la reconnaissance conjointe, dans un ordre quelconque, de plusieurs étapes du scénario (ne partageant pas d'événements communs). La syntaxe associée est :

nonordered : `"Non_ordered" "{"` *scenario* `","` *scenario*)+ `"}" ;`

L'opérateur est représenté par le symbole terminal `"Non_ordered"`. Tout comme l'opérateur `One_among`, il s'applique à plusieurs expressions de type *scenario*.

Exprimé de manière informelle, le scénario associé à un opérateur `Non_ordered` est reconnu dès que tous ses sous-scénarios ont été reconnus (en parallèle) sans partager d'événements en commun.

La sémantique opérationnelle associée à cet opérateur (cf 3.3.4) permet de considérer le scénario associé comme une simple étape du scénario global. En effet, ce scénario définit un intervalle temporel. Il commence avec l'apparition du premier événement qui démarre l'un des sous-scénarios et se termine avec l'apparition du dernier événement permettant la reconnaissance du dernier sous-

[19] équivalent (côté détection) de l'opérateur d'action `Non_ordered` précédemment introduit.

scénario.

Prenons l'exemple de scénario suivant :

```
<ENCHAIN> Non_ordered{(A0 ; B0) , (C0 ; D0)} </ENCHAIN>
```

Dans ce scénario correspondant à un opérateur `Non_ordered` appliqué à deux séquences, la reconnaissance commence avec l'apparition soit de `A0`, soit de `C0`. Elle se terminera avec l'apparition du dernier événement du dernier scénario non encore reconnu, c'est-à-dire soit `B0`, soit `D0` selon le cas.

Opérateur d'exclusion `Without`

Certaines attaques sont caractérisées par le fait qu'un ou plusieurs événements n'apparaissent pas dans le flux des événements entrants. Par exemple, le fait qu'un utilisateur exécute des programmes avec le privilège de l'administrateur sans avoir exécuté la commande *su* auparavant est certainement révélateur d'une attaque.

L'opérateur `Without` permet d'exprimer que l'on souhaite reconnaître un scénario (dit «positif») durant lequel la reconnaissance d'un scénario (dit «négatif») ne doit pas avoir lieu.

La syntaxe associée à cet opérateur est :

without : `"["` (*suite*
| *nonordered*
| *without*) `"]"` `"Without"` `"{"` *scenario* `"}"` `;`

Cet opérateur infixe est représenté par le symbole terminal `"Without"`. L'expression représentant le scénario positif est délimitée par des crochets tandis que celle représentant le scénario négatif est délimitée par des accolades.

L'expression définissant un scénario positif est un sous-ensemble de *scenario*. Elle est restreinte à *suite*, *nonordered* et *without*[20]. Ce choix tient au fait qu'un scénario positif doit impérativement être composé d'au moins deux événements distincts afin de représenter un intervalle temporel non nul (durant lequel la reconnaissance complète du scénario négatif ne doit pas intervenir). Nous avons donc exclu la possibilité d'avoir une expression correspondant aux éléments de type *evt* et *oneamong* (qui peut lui-même être réduit à un *evt*).

La reconnaissance du scénario négatif n'est déclenchée qu'après l'apparition d'un premier événement dans le scénario positif. Ensuite, si ce scénario négatif est entièrement reconnu avant la fin du scénario positif, la reconnaissance du scénario

[20] définition récursive.

positif est invalidée et l'hypothèse en cours est abandonnée. Par contre, si c'est le scénario positif qui est entièrement reconnu (sans avoir été invalidé), alors le scénario associé à l'opérateur `Without` est reconnu et l'hypothèse en cours voit son historique enrichi des événements qui ont contribué à la reconnaissance du scénario positif.

De manière informelle, on peut considérer qu'à partir du moment où le scénario négatif est activé (c'est-à-dire dès qu'il contient au moins une hypothèse en attente d'un événement), le premier des deux scénarios qui est entièrement reconnu prend l'avantage et décide si le scénario associé à l'opérateur `Without` est reconnu ou non.

Exemple :

```
<ENCHAIN> [(LoginUser ; ExecAsRoot)] Without {ExecSu} </ENCHAIN>
```

Dans cet exemple de scénario correspondant à un opérateur `Without`, l'attaque est détectée dès qu'un utilisateur utilise une commande avec des privilèges administrateur sans avoir d'abord utilisé la commande habituelle *su*. La recherche du scénario négatif (événement `ExecSu`) commence après la phase de login de l'utilisateur (événement `LoginUser`). A partir de cet instant, si un événement `ExecAsRoot` apparaît en premier, alors l'attaque est détectée puisque le scénario positif a été entièrement reconnu. Si c'est un événement `ExecSu` qui apparaît en premier, alors l'hypothèse en cours est abandonnée car on s'acheminait vers une fausse alerte[21].

Contrainte temporelle `Timeout`

Le second type de contrainte temporelle de *enchain* est représenté par le symbole non-terminal *timeout*.

timeout : `"Timeout" "(" NUMBER ")"` ;

Il permet d'exprimer explicitement une coupure dans le processus de détection en donnant une durée de vie maximale à chacune des hypothèses courantes. Le symbole terminal `NUMBER` représente le nombre de secondes associé à la contrainte. Exemple :

```
<ENCHAIN>
```

[21] En effet, si l'utilisateur connaît le mot de passe de l'administrateur, c'est qu'il est lui-même l'administrateur ou son mandataire. Sa prochaine commande (*ExecAsRoot*) n'a sûrement rien d'illégal.

```
  (A ; B ; C)
  Timeout(10)
</ENCHAIN>
```

Dans ce scénario, dès qu'un événement `A` est reconnu, on crée une nouvelle hypothèse. Sa durée de vie est fixée à 10 secondes. Si la reconnaissance d'un `B` puis d'un `C` n'intervient pas dans ce délai, alors cette hypothèse est abandonnée.

Contrainte temporelle `Mindelay`

Le troisième type de contrainte temporelle de *enchain* est représenté par le symbole non-terminal *mindelay*.

mindelay : `"Mindelay" "(" ID "," ID ")" "==" NUMBER ;`

Il permet d'exprimer qu'un intervalle de temps minimum doit s'écouler entre la reconnaissance de deux événements particuliers dans le scénario. Comme nous le verrons dans la sémantique opérationnelle (cf 3.3.7), l'ordre d'apparition relatif des deux événements importe peu car cet opérateur est symétrique. Les symboles terminaux `ID` correspondent aux noms des événements concernés par la contrainte. Le symbole terminal `NUMBER` représente le délai en secondes. Exemple :

```
<ENCHAIN>
  (A ; B ; C)
  Mindelay(B,C) == 20
</ENCHAIN>
```

dès que l'événement `B` apparaît (après l'événement `A`), d'une part, il augmente l'historique de l'hypothèse courante et, d'autre part, son horodatage fixe la borne inférieure de l'intervalle de temps minimum. A partir de cet instant, tout événement candidat à être l'événement `C` sera rejeté tant que le délai imparti n'est pas écoulé (20 secondes).

Contrainte temporelle `Maxdelay`

Le dernier type de contrainte temporelle de *enchain* est représenté par le symbole non-terminal *maxdelay*.

maxdelay : `"Maxdelay" "(" ID "," ID ")" "==" NUMBER ;`

Il permet d'exprimer que la reconnaissance entre deux événements particuliers du scénario est soumise à un intervalle de temps maximum. Comme l'opérateur

précédent, l'ordre d'apparition relatif des deux événements importe peu car il est aussi symétrique (cf 3.3.8). De même, les symboles terminaux `ID` correspondent aux noms des événements concernés par la contrainte et le symbole terminal `NUMBER` représente le délai en secondes. Exemple :

```
<ENCHAIN>
  (A ; B ; C ; D)
  Maxdelay(B,C) == 20
</ENCHAIN>
```

Dès que l'événement `B` apparaît, la reconnaissance de l'événement `C` doit avoir lieu dans les 20 secondes qui suivent. Dans le cas contraire, on abandonne l'hypothèse en cours. Cette contrainte ne s'applique qu'entre les instants représentés par la reconnaissance des événements `B` et `C`.

2.3.1.3 La sous-section `<CONTEXT>`

Après avoir défini tous les aspects temporels de la signature dans la sous-section `<ENCHAIN>`, nous allons maintenant nous intéresser à la dernière partie de la signature : l'expression des contraintes, dites «contextuelles», qui portent sur les valeurs contenues dans les instances d'événements.

Ces contraintes permettent de limiter le nombre d'événements considérés lors de la recherche d'une signature en sélectionnant seulement ceux qui sont caractéristiques de l'attaque. Elles peuvent être de type **intra-événement** (lorsqu'elles portent sur les champs d'un seul événement) ou **inter-événements** (lorsqu'elles lient entre eux les champs de plusieurs événements générés lors de la même attaque).

Nous verrons dans la partie consacrée à la sémantique opérationnelle (cf 3.4) un exemple d'algorithme mettant en oeuvre la vérification des contraintes contextuelles dans la signature.

La grammaire associée à cette dernière partie de la signature est :

context : `"<CONTEXT>"`
(*constraint*)*
`"</CONTEXT>"` ;

On peut définir autant de contraintes que nécessaire (éventuellement aucune). Lorsqu'une contrainte est évaluée, elle renvoie une valeur booléenne (`vrai` ou `faux`). Toutes les contraintes s'appliquent simultanément mais certaines ne seront évaluables que lorsque les événements concernés seront apparus dans le flux des événements entrants.

```
constraint : ID ":=" FIELD
           | FIELD "==" ID
           | FIELD "==" ( FIELD ("==" FIELD)*
                        | STRING )
           | FIELD ( " !=" | ">" | "<" | ">=" | "<=") (FIELD | STRING)
           | FIELD ( "MATCHES" STRING
                   | "IN" "[" STRING ("," STRING)* "]" );
```

Le symbole non-terminal STRING représente une chaîne de caractères constante entre double-quotes (caractère '"').

Le symbole non terminal FIELD représente l'accès à la valeur contenue dans un champ d'un événement. La notation est identique à celle définie pour le typage des événements (notation XPATH abrégée, cf 2.3.1.1) à une différence près. Le préfixe `Alert` (resp. `Network`,`System` ou `Appli`) est remplacé par le nom associé à l'instance d'événement. Par exemple, soit un événement nommé `E0` représentant une alerte IDMEF, on accède au champ qui constitue son nom officiel par la notation : `E0/Classification[0]/name`. Nous rappelons que dans le cas où un événement n'est pas en XML (c'est-à-dire dont le format n'est ni l'IDMEF, ni l'EVMEF), il faudra traduire le chemin d'accès au champ considéré pour pouvoir extraire sa valeur.

Le symbole non-terminal ID représente une variable d'unification utilisée dans les contraintes inter-événements pour fixer à une valeur commune les champs de plusieurs événements.

Tous les opérateurs utilisent les chaînes en tant que valeurs ASCII (sauf les opérateurs "<", ">", "<=", et ">=" qui effectuent des comparaisons numériques après transformation).

Contraintes intra-événement

Le typage (section `<EVENTS>`) réalise un premier niveau de filtrage sur les événements. Les contraintes qui portent sur les valeurs d'un seul événement (typé) constituent un filtrage supplémentaire.

Elles sont de deux types : soient elles constituent une comparaison à l'aide d'un opérateur entre un champ de l'événement (`FIELD`) et une constante (`STRING`), soient elles constituent une comparaison à l'aide d'un opérateur entre deux champs du même événement.

Exemples de contraintes :

- comparaison (==, !=, <, >, <=, >=) avec une constante :

```
<CONTEXT>
 E0/Target[0]/Service/dport == "21"
</CONTEXT>
```

- test de conformité (MATCHES) avec une expression régulière :

```
<CONTEXT>
 E0/Target[0]/process/args[0] MATCHES ".*/etc/passwd.*"
</CONTEXT>
```

- appartenance (IN) à une liste de valeurs (alternative) :

```
<CONTEXT>
 E0/Target[0]/Service/dport IN ["20","21","25","80"]
</CONTEXT>
```

- comparaison (==,!=,<,>,<=,>=) entre les valeurs de deux champs :

```
<CONTEXT>
 E0/Source[0]/Address/address != E0/Target[0]/Address/address
</CONTEXT>
```

Contraintes inter-événements

L'intérêt des contraintes inter-événements (portant sur les valeurs contenues dans les champs de deux ou plusieurs événements) réside dans la possibilité de sélectionner dans le flux des événements ceux qui résultent de l'observation de la même attaque, afin d'éviter au maximum les faux positifs. C'est le coeur même du processus de corrélation.

Cela permet d'exprimer, par exemple, que tous les événements générés lors de l'attaque contiennent la même adresse IP cible ou encore que c'est le même utilisateur qui a exécuté les commandes incriminées.

Exemples de contraintes :

- déclaration en plusieurs fois d'une relation d'égalité sur la valeur des champs de plusieurs événements (grâce à une variable d'unification explicite) :

```
<CONTEXT>
 X := E0/Target[0]/Node/Address/address
 E1/Target[0]/Node/Address/address == X
 E2/Target[0]/Node/Address/address == X
</CONTEXT>
```

- déclaration en une seule fois d'une relation d'égalité sur la valeur des champs de plusieurs événements (trois ou plus) :

```
<CONTEXT>
 E0/snare/user/uid == E1/snare/user/uid == E2/snare/user/uid
</CONTEXT>
```

- comparaison (==,!=,<,>,<=,>=) entre les valeurs des champs de deux événements distincts :

```
<CONTEXT>
 E0/packet/ether/ip/tcp/dport != E1/packet/ether/ip/tcp/sport
</CONTEXT>
```

2.3.2 La section <CONFIRM>

Après avoir détaillé l'expression d'une signature dans le langage ADeLe, nous passons à la phase qui suit la détection d'une attaque où l'on essaie de confirmer et d'améliorer le diagnostic de détection.

Lorsqu'une attaque a été détectée, cela signifie que tous les événements caractéristiques de l'attaque ont été observés et qu'ils vérifiaient les contraintes temporelles et contextuelles. Dans l'hypothèse où cette signature ne génère pas de faux positifs, on considère que l'attaque s'est bien produite. Toutefois, certaines signatures ne peuvent pas déterminer (sur la base des événements observés) si l'attaque a réussi ou si c'était une tentative qui a échoué.

Nous proposons dans cette section d'exprimer, à l'aide d'un langage de scripts utilisant des fonctions prédéfinies, des tests sur la réussite ou l'échec de l'attaque afin d'affiner le diagnostic de détection. Cela permet notamment de mieux gérer les priorités des alertes notifiées à l'opérateur (en affectant un indice de sévérité moindre à celles qui ont échoué ou dont les conséquences sont les moins graves).

Le format IDMEF [CD03], dans sa version 1.0, prévoit des champs facultatifs pour ce type d'informations dans les attributs XML de la classe `Impact` :

- `Alert/Assessment/Impact@severity` représente une estimation de la gravité de l'alerte. Il peut prendre les valeurs `'low'`, `'medium'` ou `'high'` ;
- `Alert/Assessment/Impact@completion` détermine si la tentative d'attaque a réussi ou échoué. Il peut prendre les valeurs `'failed'` ou `'succeeded'` ;
- `Alert/Assessment/Impact@type` représente le type de tentative d'attaque effectué. Il désigne respectivement l'obtention des privilèges de l'administrateur (`'admin'`), un déni de service (`'dos'`), une action sur un fichier (`'file'`), une phase de reconnaissance avant attaque (`'recon'`), l'obtention des privilèges d'un utilisateur (`'user'`) ou toute autre catégorie non répertoriée (`'other'`).

Pour obtenir ces informations complémentaires, il faut entreprendre des actions sur les machines surveillées[22], localement ou à distance, pour vérifier activement si les conséquences normales d'une attaque réussie sont bien présentes. Lorsqu'elle est possible, cette vérification peut être exprimée grâce à des scripts utilisant des fonctions booléennes prédéfinies. Le résultat de la vérification est alors conservé dans des variables afin d'être utilisé dans la phase suivante de construction de l'alerte (section `<REPORT>`). Comme pour la balise `<CODE>` de la

[22]dans le cas où les commandes de vérification doivent s'exécuter sur la machine cible, il faut envisager de déployer des agents. Toutefois, la réponse d'un agent installé sur une machine compromise (avec des privilèges administrateur) est sujette à caution.

partie <EXPLOIT> où le code peut être exprimé dans n'importe quel langage, la grammaire de cette section ne détaille pas le langage de scripts qui est utilisé :

confirm : "<CONFIRM>"
(TEXTE) ?
"</CONFIRM>" ;

Par exemple, dans le cas d'une attaque en déni de service de type "Ping of Death"[23] réussie, la machine attaquée affiche un écran bleu et ne répond à aucune sollicitation extérieure. La signature correspondante ne fait que rechercher l'apparition de paquets ICMP particuliers sur le réseau. Elle ne permet pas de savoir si la pile IP de la machine cible était vulnérable à cette attaque. Si l'on veut connaître l'issue de cette attaque (réussite ou échec), il faut donc tester à distance si la machine attaquée est hors-service (par exemple avec des requêtes ICMP de type *Echo request*). Cet exemple peut être traduit par la spécification suivante :

```
<CONFIRM>
  String severity, completion, type ;
  IF Unreachable_Machine(E0/Target[0]/Address/address){
    severity := "high";
    completion := "succeeded";
    type := "dos";
  } ELSE{
    severity := "low";
    completion := "failed";
    type := "dos";
  }
  # on exporte les valeurs pour le REPORT
  Report(severity,completion,type);
</CONFIRM>
```

La fonction booléenne prédéfinie `Unreachable_Machine()` prend en entrée l'adresse IP de la machine cible de l'attaque (valeur contenue dans un champ de l'un des événements qui ont conduit à la détection). Elle renvoie vrai si la machine n'est plus joignable (donc vraisemblablement hors-service) et faux sinon. Le résultat permet d'ajuster la valeur des trois variables `severity`, `completion` et `type`. La procédure `Report()`, appelée avec nos trois variables en paramètre, est chargée d'exporter leurs valeurs pour une utilisation ultérieure dans la section <REPORT>.

[23] envoi de paquets de type ICMP de taille anormalement grande (CVE-1999-0128).

On peut définir une librairie de fonctions qui seront ensuite réutilisées dans plusieurs descriptions.

Par exemple, certaines attaques réseaux sont destinées à la pile IP d'un système d'exploitation (OS) particulier. Si l'OS de la machine cible ne correspond pas, alors l'attaque a échoué. Une fonction `OsTypeMatch(`*adresse_IP*, *type_OS*`)` permettrait, en interrogeant une base de données qui référence le(s) type(s) d'OS installés sur chaque machine, de savoir si la machine était vulnérable à l'attaque.

De même, si une attaque a pour effet d'installer une porte dérobée (*backdoor*) en écoute sur un port TCP inhabituel et fixé d'avance, une fonction `Backdoor-ListeningTCP(`*adresse IP*, *port*`)` permettrait, en tentant d'établir une connexion sur ce port, de confirmer la réussite de l'attaque.

Dans le cas où la détection se fait en mode *off-line* à partir d'événements ou d'alertes archivés, cette phase de vérification active ne doit pas avoir lieu car le résultat risque d'être erroné. En effet, l'état des systèmes concernés par l'attaque a certainement changé entre l'heure réelle de l'attaque et l'heure de la détection. De plus, l'analyseur qui effectue la détection n'a peut-être plus aucune connectivité réseau avec les machines surveillées.

2.3.3 La section `<REPORT>`

Dans la troisième et dernière section de la partie `<DETECTION>`, nous rassemblons toutes les informations permettant de construire une alerte spécifique à l'attaque qui a été détectée.

Cette alerte, au format standard IDMEF, sera envoyée au manager. Sa spécification consiste à donner l'intégralité des champs XML à créer ainsi que les valeurs associées. Comme dans les sections précédentes, nous utilisons une représentation XPATH abrégée pour désigner l'accès à un champ XML de l'alerte.

La syntaxe associée est donnée par la grammaire suivante :

```
report :  "<REPORT>"
          ( FIELD ":=" ( STRING
                       | FUNCTION
                       | ( FIELD ("|" FIELD)*
                       )
          )+
          "</REPORT>" ;
```

Le symbole non-terminal FIELD à gauche de l'affectation est toujours préfixé par "Alert"[24] (ex :`Alert/Classification[0]/name`) car il désigne un champ de

[24]L'encapsulation de ces champs dans une balise "<IDMEF-Message>" globale est implicite.

l'alerte en cours de construction. Par contre, les occurences du symbole FIELD qui apparaissent à droite de l'affectation commencent par un nom d'événement déclaré dans la section `<EVENTS>` (ex : `E0/Node/name`) et représentent la valeur ASCII associée au champ XML correspondant.

Le symbole non-terminal STRING désigne une chaîne de caractères constante entourée de doubles quotes (ex : "NFS_Mount").

Le symbole FUNCTION désigne un appel de fonction, éventuellement avec des paramètres (ex : *NewAlertID()*).

La règle *report* constitue simplement une répétition de lignes, dont chacune conduira à l'affectation d'une valeur ASCII à un champ XML (noeud textuel ou attribut) de l'alerte. Ces valeurs proviennent de trois types de déclarations différentes :

- **valeurs constantes**
  Certains champs ont une valeur connue à l'avance comme, par exemple, le nom associé à l'attaque.

  Exemples :
  ```
  Alert/Classification[0]/origin := "ADeLe"
  Alert/Classification[0]/name   := "NFS_Mount"
  ```

- **valeurs générées par des fonctions**
  Certains champs nécessitent le calcul d'une valeur à l'exécution par une fonction. `NewAlertId` génère un numéro d'alerte unique. `AnalyzerName` et `AnalyzerIP` identifient l'analyseur courant (resp. nom et adresse IP). `NewTime` et `NewNtpStamp` renvoient la date courante. `Value` permet d'obtenir la valeur d'une variable exportée par la section `<CONFIRM>` ou bien une valeur par défaut.

  Exemples :
  ```
  Alert@ident                          := NewAlertid()
  Alert/Analyzer@ident                 := AnalyzerName()
  Alert/Analyzer/Node/Address/address := AnalyzerIP()
  Alert/CreateTime@ntpstamp            := NewNtpStamp()
  Alert/CreateTime                     := NewTime()
  Alert/Assessment/Impact@severity     := Value(severity,"high")
  Alert/Assessment/Impact@completion   := Value(completion,"succeeded")
  Alert/Assessment/Impact@type         := Value(impact,"user")
  ```

- **valeurs extraites des événements**
  Certains champs doivent utiliser des valeurs contenues dans les instances

d'événements qui ont permis la détection. Ces valeurs représentent des détails intéressants de l'attaque tels que l'adresse IP de l'attaquant, le nom de la machine attaquée ou encore l'horodatage contenu dans les événements.

Exemples :

```
Alert/DetectTime@ntpstamp            := E6/DetectTime@ntpstamp
Alert/DetectTime                     := E6/DetectTime
Alert/Source[0]/Node/Address/address := E6/Source[0]/Node/Address/address
Alert/Target[0]/Node/Address/address := E0/Target[0]/Node/Address/address
```

Comme le scénario temporel inclus dans la signature peut contenir des alternatives (opérateur de type One_among), il se peut qu'une instance d'événement référencée dans la signature n'existe pas. Par exemple, si l'on a déclaré le scénario suivant :

```
<ENCHAIN> (E1 ; Non_ordered{E2 , E3}) </ENCHAIN>
```

on ne sait pas a priori si la détection a porté sur l'événement `E2` ou sur l'événement `E3`. Cela pose donc un problème si on doit extraire la valeur d'un champ de l'un de ces deux événements car on ne sait pas lequel existe au moment de la construction de l'alerte.

Nous avons donc prévu une notation spéciale pour le cas de l'alternative. Plusieurs champs (appartenant à des événements déclarés dans des branches distinctes d'un même opérateur `One_among`) peuvent être déclarés dans la même affectation, séparés par le symbole '`|`'. A l'exécution, le bon événement sera choisi et la valeur sera extraite du champ correspondant. Exemple :

```
<REPORT>
  Alert/Target[0]/Node/name := E2/Target[0]/Node/name
                             | E3/Target[0]/Node/name
  ...
</REPORT>
```

2.4 La partie <RESPONSE>

La troisième et dernière partie d'une description en ADeLe se place également du point de vue du défenseur. Elle donne la possibilité d'exprimer les éventuelles contre-mesures qui peuvent être prises automatiquement en cas de détection de l'attaque, pour la stoppper ou éviter qu'elle ne se reproduise.

Mis à part l'article [MM01] qui pose les fondements du langage ADeLe, nous n'avons connaissance que d'une seule publication ([GD02]) consacrée à la réaction automatique aux attaques.

Cette fonctionnalité peut s'avérer dangereuse et doit être utilisée avec prudence car elle peut facilement être détournée de son objectif initial par un attaquant pour provoquer des attaques en déni de service envers le réseau surveillé. En effet, de nombreuses attaques réseau comportent une falsification (*spoofing*) de l'adresse IP censée être à l'origine de l'attaque.

Par exemple, si un attaquant usurpe des adresses IP assignées aux serveurs DNS[25] racines d'Internet et que l'IDS réagit automatiquement en bloquant ces adresses au niveau du pare-feu d'entrée du réseau, alors la plupart des communications avec l'extérieur seront rompues car il sera désormais impossible de faire de la résolution de noms.

Toutefois, nous pensons qu'une réaction automatique, même pour un nombre restreint d'attaques, peut être utile. On peut prendre quelques précautions pour limiter le risque de déni de service. Par exemple, les restrictions peuvent être appliquées seulement à des adresses IP externes (avec maintien d'une liste blanche pour ne pas bannir des adresses cruciales telles que les serveurs DNS racines d'Internet).

On peut aussi n'autoriser une réaction automatique que pour les types d'attaque où il ne peut pas y avoir de *spoofing*. Dans tous les autres cas où il y a un risque de déni de service, il est souhaitable que ce soit l'opérateur humain qui prenne la décision finale de réagir à l'attaque. Cela signifie que le module de réaction lui propose les actions à entreprendre. D'autres raisons (extérieures à la description de l'attaque) peuvent également justifier l'absence de réaction automatique. Par exemple, la loi en vigueur dans le pays où a lieu la détection peut considérer que les contre-mesures envisagées sont elles-mêmes illégales.

Comme pour la section `<CONFIRM>`, la syntaxe interne de cette partie est libre :

response : "<RESPONSE>"
(TEXTE) ?
"</RESPONSE>" ;

Nous proposons de spécifier ces actions à l'aide de scripts pré-définis (dont les paramètres sont issus des valeurs contenues dans les événements qui ont permis la détection). Au cas où la réaction doit être réalisée manuellement par l'opérateur, il est possible d'écrire des commentaires (expliquant ce qu'il faudrait faire) qui seront présentés à l'opérateur en parallèle de l'alerte.

Nous donnons quelques indications de scripts qui peuvent être utilisés pour

[25] *Domain Name Server*

arrêter l'attaque en cours ou empêcher qu'elle ne recommence.

Afin de stopper l'attaque en cours, on peut prendre les mesures suivantes :

- couper une communication TCP établie.
  `Reset_TCP_connection(src_ip, src_port, dst_ip, dst_port)` déclenche l'émission de deux paquets TCP Reset[26] (un vers chaque extrémité de la connexion).
- tuer un (ou plusieurs) processus.
  `Kill_Process(dst_ip, user_name, proc_id | "ALL")` permet de tuer un processus particulier ou tous ceux d'un utilisateur ciblé.
- éteindre une machine.
  `Shutdown_Machine(dst_ip)` peut éteindre une machine à distance dans le cas où l'intégrité de ses données est menacée.

Une méthode (à court terme) pour empêcher que l'attaque ne se reproduise, en attendant de pouvoir corriger la vulnérabilité utilisée, consiste à supprimer le service incriminé :

`Close_Service(dst_ip, protocol, dst_port)` supprime le service lié au port utilisé par l'attaque (modification de la configuration du *daemon* "inetd" sur la plupart des Unix, désactivation d'un service dans le monde Windows).

Lorsque la méthode précédente n'est pas possible, par exemple dans un contexte commercial où la contrainte de disponibilité du service l'emporte sur le risque d'attaque, on peut interdire temporairement à la source de l'attaque l'accès au système d'information :

`Firewall_Black_List(src_ip)` ajoute une adresse IP à une liste noire maintenue par un pare-feu. On peut citer l'outil SnortSam[27] (plugin pour l'IDS Snort) qui offre cette fonctionnalité.

Pour pouvoir exécuter des actions arbitraires (autres que l'un des scripts prédéfinis), on prévoit l'exécution de scripts externes : `Script_Exec(script_name)`.

Lorsque la détection de l'alerte a été réalisée en mode *off-line*, la phase de réaction ne doit pas être prise en compte.

Exemple tiré de l'attaque NFS_Mount :

```
<RESPONSE>
  Reset_TCP_connection( E6/Source[0]/Node/Address/address,
                        E6/Source[0]/Node/Service/port,
                        E6/Target[0]/Node/Address/address,
                        E6/Target[0]/Node/Service/port) ;
</RESPONSE>
```

[26] il faut donc attendre le prochain paquet de la connexion pour en extraire des numéros de séquence TCP valides.

[27] http://www.snortsam.net

Chapitre 3

Sémantique opérationnelle de la détection

Le chapitre précédent nous a présenté en détail la syntaxe utilisée par le langage ADeLe et, de manière informelle, la sémantique associée (c'est-à-dire l'interprétation de ce que l'on peut écrire). Afin que ce langage ne reste pas une construction abstraite, nous avons choisi de proposer une mise en œuvre de la partie détection (chapitre 4), ce qui suppose une définition stricte de sa sémantique.

Dans ce chapitre, nous donnons une sémantique opérationnelle à la partie détection du langage ADeLe. Cela signifie que l'interprétation est entièrement définie par un algorithme qui permet d'obtenir des résultats conformes à la sémantique. Ainsi, toute autre mise en œuvre qui respecte la sémantique originelle doit fournir les mêmes résultats à partir des mêmes données, quels que soient les algorithmes utilisés.

Ce chapitre est organisé de la façon suivante. Dans la section 3.1, nous présentons rapidement les pistes suivies pour trouver un formalisme permettant d'exprimer simplement la sémantique opérationnelle associée à la détection.

Dans la section 3.2, nous définissons les éléments de base de notre formalisme utilisant des automates à états finis (types abstraits de données utilisés, notations pour les propriétés temporelles).

Dans la section 3.3, nous présentons la sémantique associée aux différents opérateurs (ou contraintes) de corrélation temporelle entre événements, en définissant les propriétés des sous-automates équivalents.

Dans la section 3.4, nous donnons la sémantique associée aux différentes contraintes qui définissent la corrélation contextuelle, c'est-à-dire les contraintes qui portent sur les valeurs contenues dans les événements.

Dans la section 3.5, nous présentons le principe de fonctionnement de l'algo-

rithme abstrait utilisé pour détecter une attaque. Il s'appuie sur un automate à états finis pour traiter un flux d'événements et y rechercher la signature correspondante.

Dans la section 3.6, nous abordons les problèmes d'explosion combinatoire inhérents à notre approche ainsi que des solutions permettant de les réduire.

3.1 Choix du formalisme

Plusieurs pistes ont été suivies pour choisir un formalisme existant permettant d'exprimer facilement la sémantique opérationnelle associée à la partie détection du langage.

Statecharts

Nous avons étudié la possibilité d'utiliser le formalisme des Statecharts [Har87] pour l'appliquer à notre problème de reconnaissance d'une signature dans un flux d'événements. Il comporte des concepts intéressants tels que le parallélisme ou bien la composition hiérarchique. Malheureusement, il ne permet pas d'exprimer autre chose qu'une information binaire lorsque l'on passe d'un état au suivant (équivalent aux jetons des réseaux de Petri classiques). On ne peut donc pas accumuler de l'information au cours de la progression dans l'automate résultant, ce qui est nécessaire dans notre contexte de recherche exhaustive des instances de reconnaissance d'une signature dans un flux d'événements.

LOTOS

Nous avons également envisagé l'utilisation de LOTOS [ISO87], qui est un langage basé sur l'algèbre de processus (CCS & CSP) ainsi que les types de données abstraits (ACT-ONE). Les opérateurs fournis par ce langage sont intéressants (séquence, choix (non-)déterministe, parallélisme ...). Malheureusement, la syntaxe utilisée est complexe et certains opérateurs sont difficiles à mettre en œuvre dans la pratique.

Colored Petri Automaton (CPA)

Comme dans l'approche proposée par Kumar et Spafford [**?**], nous avons envisagé l'utilisation d'un modèle basé sur les réseaux de Petri (RdP).

Dans notre cas, la signature serait transformée en un réseau de Petri équivalent où les transitions sont tirées sur l'apparition des événements associés (RdP *synchronisés* [MPS78]) et où des gardes peuvent être exprimées pour représenter les contraintes entre événements.

L'utilisation d'un jeton classique ne permet pas de transporter un contexte dans le réseau de Petri. Nous aurions donc dû utiliser des propriétés des RdP

dits *colorés* [Jen81] qui autorisent que les jetons comportent plusieurs attributs (appelés couleurs) afin de conserver et transmettre des attributs d'événements.

Le problème principal vient du fait que nous voulons la liste **exhaustive** des occurences de la signature. Etant donné un flux d'événements et une signature, nous ne voulons pas seulement **un** n-uplet d'événements qui correspond à la reconnaissance de la signature mais **tous** les n-uplets qui correspondent. L'utilisation du formalisme des réseaux de Petri (même colorés et synchronisés) devient alors assez lourde. En effet, lorsqu'un événement permet de tirer une transition, on doit créer un nouveau jeton (reflètant la progression dans la reconnaissance de la signature) dans la place suivante ET conserver l'ancien dans la place initiale, afin de conserver toutes les hypothèses intermédiaires. On doit alors ajouter une transition supplémentaire à chaque état pour qu'il boucle sur lui-même. De plus, il existe des cas (opérateur **One_among**) où un seul événement doit permettre le franchissement simultané de plusieurs transitions, ce qui semble difficile à exprimer dans le formalisme des réseaux de Petri.

Automates à états finis à «exécution parallèle»

Finalement, nous avons choisi comme base de la sémantique de la signature dans ADeLe une solution inspirée des RdP mais utilisant des automates à états finis supportant des «exécutions parallèles».

La signature est traduite par compilation en un automate à états finis permettant d'en assurer la détection, représenté par un graphe dirigé acyclique. Cet automate sert de modèle à la détection et nous permet de représenter l'état de reconnaissance de la signature. Il garantit, par construction des états, des transitions et des conditions associées aux transitions, qu'une hypothèse qui progresse vers l'état d'acceptation respecte les enchaînements temporels et les contraintes contextuelles définis en ADeLe.

Comme les jetons des réseaux de Petri colorés, chaque hypothèse (contenue dans un état) transporte avec elle un ensemble d'informations qui représentent une reconnaissance partielle de la signature. Les hypothèses progressent en parallèle au sein de l'automate.

Comme pour les RdP synchronisés, les transitions ne sont franchies que sur l'apparition d'événements particuliers (issus du filtrage/typage des événements concrets remontés par les capteurs ou les sondes). Chaque événement traité par notre automate est fourni simultanément à toutes les transitions concernées (c'est-à-dire qui attendent ce type d'événement). Les contraintes contextuelles du langage sont traduites en gardes (portées par les transitions) qui assurent qu'un événement remplit toutes les conditions requises avant de pouvoir faire progresser une hypothèse.

Lorsqu'une hypothèse franchit une transition, son historique est augmenté avec l'événement courant. On garde également une copie de l'ancienne hypothèse

dans l'état d'origine afin de conserver toutes les hypothèses intermédiaires. Cela nous permet d'opérer une **recherche exhaustive** de toutes les occurences de reconnaissance de la signature en analysant le flux d'événements en une seule passe.

Toute hypothèse qui atteint l'état d'acceptation de l'automate correspond à une occurence distincte de reconnaissance de la signature. On émet alors une alerte basée sur le n-uplet des événements qui ont fait progresser cette hypothèse.

On note que la construction et l'initialisation de l'automate doivent être réalisées avant de pouvoir analyser le flux d'événements.

3.2 Les bases du formalisme

3.2.1 Types abstraits de données utilisés pour les automates

Nous avons défini un certain nombre de types de données adaptés à la résolution algorithmique de notre problème de la détection. Ils rassemblent l'ensemble des informations nécessaires à la construction de notre modèle à base d'automates de reconnaissance. Ils sont couramment utilisés dans l'algorithme qui définit la sémantique opérationnelle de la détection dans le langage ADeLe.

Automate
Un automate modélise la reconnaissance de la signature associée à la détection d'une attaque particulière. Il relie entre eux un ensemble d'états (*etats*) grâce à un ensemble de transitions (*transitions*). Il comporte également un état de départ (*init*) utilisé pour le marquage initial. Il peut éventuellement comporter une durée de vie maximale (*timer*) pour les hypothèses.

Automate = (etats : **ensemble** d'Etat,
transitions : **ensemble** de Transition,
init : Etat,
timer : entier) ;

Log
Le type *Log* représente un flux d'événements entrants (issus des capteurs et des sondes). Chacun des événéments de ce flux est du type prédéfini *EvenementBrut* qui représente tous les types de format (IDMEF, EVMEF, PACKET, SNARE, BSM ...).

Log = **liste** d'EvenementBrut ;

Evenement
Un événement filtré est le résultat de la fonction de filtrage (associée à l'attaque) appliquée à un événement concret (de type *EvenementBrut*) du flux des événements entrants. Il se caractérise par un type (*type*), un horodatage (*heure*) et un ensemble de paires (*paires*) qui représentent les champs utiles[1] extraits de l'événement concret originel. Ce sont ces événements filtrés qui vont alimenter notre automate.

Evenement = (type : chaine,
heure : entier,
paires : **ensemble** de Paire) ;
Paire = (nom : chaine, valeur : chaine) ;

Etat
Un état comporte un type associé (*type*). C'est ce type qui détermine les actions effectuées sur une hypothèse lorsqu'elle arrive dans cet état. Un état comporte également deux hashtables pouvant contenir des hypothèses. La première hashtable (*courant*) représente les hypothèses présentes dans cet état au pas de simulation courant (rangées par état de provenance). La seconde hashtable (*futur*) représente les nouvelles hypothèses (rangées par état de provenance) qui ont été créées dans l'état courant au pas de simulation courant et qui ne devront être prises en compte qu'au pas de simulation suivant. Chaque état dispose également de la liste de ses transitions sortantes (*listeTrans*).

Etat = (type : TypeEtat,
courant : **hashtable** de (Etat, **ensemble** de Hypothese),
futur : **hashtable** de (Etat, **ensemble** de Hypothese),
listeTrans : **liste** de Transition) ;
TypeEtat = NORMAL | DISTRIB | MERGE | MULTI | MONO | WITHOUT
| SYNC | FINAL ;

Transition
Une transition relie un état origine (*orig*) à un état destination (*dest*). Elle comporte un type associé (*type*) qui correspond au type d'événement attendu par cette transition (sauf si c'est une ε-transition représentée par une chaîne vide). Elle peut contenir également un nom (*nomEvt*) qui sera associé à l'événement qui a permis de franchir la transition (sauf si instance anonyme). Une transition peut également contenir un ensemble de gardes (*gardes*) qui doivent toutes être

[1] c'est-à-dire les champs sur lesquels portent une contrainte (`<CONTEXT>`) ou qui sont utilisés pour construire l'alerte (`<REPORT>`)

satisfaites pour qu'une hypothèse franchisse la transition. Dans la branche positive d'un opérateur Without, certaines transitions (issues des premiers états de type `NORMAL` atteignables depuis le début de la branche) doivent déclencher un (ou plusieurs) scénario(s) inhibiteur(s). On dispose de la liste des états initiaux de ces scénarios (*inhib*). Certaines transitions comportent un délai (*timer*) qui donne une durée de vie à chacune des hypothèses qui les franchissent.

Transition = (orig : Etat,
dest : Etat,
type : chaine,
nomEvt : chaine,
gardes : **ensemble** de Garde,
inhib : **liste** d'Etat,
timer : entier) ;

Garde

Une garde est une contrainte portée par une transition. Elle s'applique aux valeurs contenues dans l'événement courant et/ou aux informations accumulées dans l'hypothèse courante. Si toutes les gardes d'une transition sont vérifiées, alors le franchissement de la transition par l'hypothèse est autorisé.

Garde = (type : Intra → champ : chaine, op : Operateur, val : Valeur
| Intra2 → champ1 : chaine, op : Operateur, champ2 : chaine
| Unif → var : chaine, op : Operateur, champ1 : chaine , champ2 : chaine
| Unif2 → var1 : chaine, op : Operateur, delai : entier, var2 : chaine) ;
| Unif3 → var : chaine, champ : chaine, var1 : chaine, op : Operateur, var2 : chaine) ;
Operateur = EQ | DIFF | INF | SUP | INFEQ | SUPEQ | MATCHES | IN
| MINDELAY | MAXDELAY ;

Hypothese

Chaque hypothèse distincte, contenue dans un état, représente une reconnaissance partielle du scénario. Elle contient toutes les informations accumulées au cours de son parcours dans l'automate de reconnaissance. Le champ *depuisEtat* référence l'état qui a poussé l'hypothèse dans l'état courant. Cette information est utile pour accéder aux ensembles d'hypothèses (rangées par état de provenance) d'un état de type `NORMAL` ou `MERGE`. Deux listes d'entiers (*sync* et *with*) servent, dans les niveaux d'imbrications des opérateurs `Non_ordered` et `Without`, à repérer les hypothèses qui sont issues de la même génération. Chaque hypothèse maintient un historique des événements (*histo*) qui l'ont fait progresser. Cette structure est

une liste de listes d'Evenement car on gère plusieurs niveaux d'imbrication de l'opérateur `Non_ordered` à l'aide d'empilement d'historiques locaux. On maintient en parallèle une association (*alias*) entre le nom d'instance porté par une transition et l'événement de l'historique qui a permis de la franchir. Pour la vérification des contraintes croisées, chaque hypothèse comporte une hashtable de variables d'environnement (*env*). Le champ *timeout* (s'il est non nul) représente la date de péremption de l'hypothèse.

Hypothese = (depuisEtat : Etat,
sync : **liste** d'entier,
with : **liste** d'entier,
histo : **liste** de Historique,
alias : **hashtable** de (chaine, Evenement),
env : **hashtable** de (chaine, chaine),
timeout : entier) ;

Historique
Un historique représente la liste ordonnée des événements qui ont fait progresser l'hypothèse. Si l'hypothèse est au sein d'une ou plusieurs imbrications de sous-scénarios représentant des opérateurs `Non_ordered`, elle comporte plusieurs niveaux d'historiques. L'historique de plus haut niveau (appelé historique local) représente alors la suite des événements survenus depuis l'entrée dans le sous-scénario correspondant au `Non_ordered` le plus interne. Lorsqu'une hypothèse sort d'un niveau d'imbrication (fusion de plusieurs hypothèses partielles), elle perd son historique de plus haut niveau mais son nouvel historique local a augmenté (fusion des historiques locaux de chacune des hypothèses partielles). Lorsqu'une hypothèse ne comporte qu'un seul niveau d'historique (ex : dans l'état d'acceptation), c'est un historique global qui représente la suite des événements survenus depuis le début de la reconnaissance du scénario.

Historique = **liste** de Evenement ;

3.2.2 Notations utilisées pour les propriétés temporelles

Afin de pouvoir exprimer des propriétés temporelles sur les scénarios[2], nous définissons quelques notations utilisées par la suite.

[2]nous nous référons ici à la règle *scenario* de la grammaire, qui constitue la première expression obligatoire de la section `<ENCHAIN>` (cf 2.3.1.2). Elle définit l'ordre partiel d'apparition des événements.

Nous précisons que tous les événements (instance nommée ou instance anonyme représentée par son type) invoqués dans un scénario doivent être des occurences distinctes (issus d'événements concrets apparaissant à des indices différents dans le flux d'événements analysé).

Un scénario est soit réduit à un seul événement, soit constitué de plusieurs événements dont on définit les contraintes d'enchaînement temporel à l'aide des opérateurs suivants : séquence (" ;"), disjonction ("One_among"), conjonction ("Non_ordered") et exclusion ("Without").

Soit S un scénario basé sur la reconnaissance de n événements E_i (avec $i = 1..n$ et $n \geq 1$) datés respectivement $t(E_i)$, alors l'intervalle temporel qui le représente est défini par une borne inférieure $t_{inf}(S)$ et une borne supérieure $t_{sup}(S)$. On pose alors que :

$$t_{inf}(S) = \min_{i=1..n}(t(E_i)) \tag{3.1}$$

$$t_{sup}(S) = \max_{i=1..n}(t(E_i)) \tag{3.2}$$

Le mécanisme de détection d'intrusions que nous considérons est basé sur des événements ponctuels. On pose alors que l'instant qui représente le moment de la reconnaissance du scénario S est égal à la borne supérieure de l'intervalle qu'il représente, c'est-à-dire :

$$t(S) = t_{sup}(S) \tag{3.3}$$

Lorsque $n = 1$, c'est-à-dire lorsque S est un scénario réduit à un seul événement E daté $t(E)$, l'intervalle temporel qui le représente est réduit à un point. En effet, par les propriétés des fonctions min et max, on déduit des équations 3.1 et 3.2 que :

$$t_{inf}(S) = t_{sup}(S) = t(E)$$

3.3 Sémantique de la corrélation temporelle

Dans la syntaxe, un scénario est défini comme la composition récursive de sous-scénarios à l'aide d'opérateurs (dont le sous-scénario minimal est un événement simple). Dans la sémantique opérationnelle, nous traduisons chaque sous-scénario en un sous-automate de reconnaissance équivalent qui représente une composition récursive de sous-automates (dont le sous-automate minimal reconnait un seul événement).

Dans les sections suivantes, nous présentons la sémantique opérationnelle associée à chacune des constructions syntaxiques de la section <ENCHAIN> (qui définit les contraintes temporelles de la signature).

3.3.1 Reconnaissance d'un seul événement

L'élément de base de la reconnaissance est un sous-automate qui ne reconnait qu'un seul événement. Il est modélisé grâce à deux états reliés par une transition (cf figure 3.1).

Il comporte un état de début de type `NORMAL`, qui est actif lorsqu'il contient une ou plusieurs hypothèses. Cet état comporte une seule transition de sortie qui attend l'apparition d'un événement.

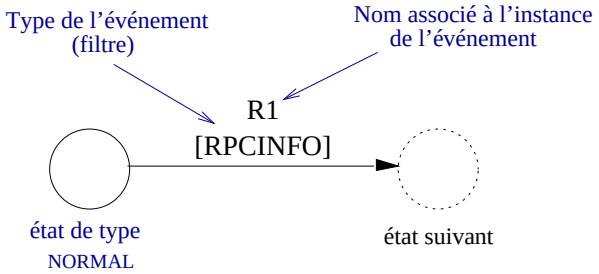

FIG. 3.1 – Sous-automate équivalent à la reconnaissance d'un événement

La transition est réceptive à un seul type d'événement (`RPCINFO` dans l'exemple). Lorsqu'un événement adéquat apparaît dans le flux des événements analysés, la transition devient franchissable. Chacune des hypothèses qui est dans l'état d'origine est alors candidate au franchissement. Si la transition comporte des gardes (c'est-à-dire des conditions logiques à vérifier), alors elles doivent toutes être vérifiées pour autoriser le franchissement d'une hypothèse.

Lorsque la transition est tirée, l'événement a été reconnu : l'hypothèse est alors modifiée puis transmise à l'état suivant. La modification porte sur la mise à jour des attributs[3] *histo* et *alias*. Ils représentent respectivement l'historique cumulé des événements reconnus jusqu'à présent par l'hypothèse et la liste des correspondances nom/instance d'événement. Une copie de l'ancienne hypothèse reste dans l'état d'origine afin de conserver les reconnaissances intermédiaires (contexte de recherche exhaustive).

3.3.2 Opérateur de séquence " ;"

En ADeLE, la séquence est représentée par une succession de sous-scénarios qui correspondent à la reconnaissance soit d'un événement, soit d'une répétition, soit de l'un des trois opérateurs (`Non_ordered`, `One_among`, `Without`).

Comme nous l'avons indiqué dans le détail de la syntaxe (cf 2.3.1.2, p. 47), la séquence définit un ordre **relatif** pour la reconnaissance de plusieurs sous-

[3]sur certaines transitions, l'attribut *timeout* peut également être modifié

scénarios. Chacun des éléments à reconnaître dans une séquence peut être séparé du suivant par un nombre arbitraire d'événements.

Soit S un scénario défini par "($<I_1>$; ... ; $<I_n>$)" où les I_i représentent des sous-scénarios tels que décrits précédemment (avec $i = 1..n$ et $n > 1$), la séquence est définie par la propriété suivante :

$$\forall\, j, 1 \leq j < n,\ t_{sup}(I_j) \leq t_{inf}(I_{j+1})$$

Cela signifie qu'un sous-scénario doit être entièrement reconnu avant que l'on commence à reconnaître le suivant. Dans le cas où l'égalité est stricte, les horodatages sont identiques mais ils sont issus d'événements concrets ayant des indices croissants dans le flux des événements d'entrée.

En termes d'automates, une séquence de plusieurs sous-scénarios est traduite par une fusion de l'état **après** reconnaissance d'un sous-scénario avec l'état **avant** reconnaissance du sous-scénario qui le suit.

N.B. : la répétition (ex : `"(RPCINFO^3)"`) est interprétée comme une séquence d'événements (`"(RPCINFO ; RPCINFO ; RPCINFO)"`) dont les instances sont anonymes. Elle ne donne donc lieu, en termes d'automates, à aucune construction particulière autre que la séquence.

Exemple

La figure 3.2 représente un exemple de scénario (constitué d'une séquence de trois événements) ainsi que l'automate équivalent. Les conventions graphiques utilisées dans cette figure sont les suivantes :

- un cercle simple (avec un numéro) représente un état de type `NORMAL` qui peut contenir une ou plusieurs hypothèses (qui sont autant de reconnaissances partielles du scénario) ;
- une flèche en trait plein (avec un nom d'événement et un type entre crochets) représente une transition déclenchable sur occurence d'un événement ;
- un cercle double (avec un numéro) représente un état de type `FINAL`, c'est-à-dire l'état d'acceptation de l'automate, qui symbolise la reconnaissance complète du scénario.

Le tableau 3.1 représente un exemple de la reconnaissance du scénario précédent (figure 3.2). Nous notons ai, bi et ci les événements concrets typés respectivement `ALPHA`, `BETA` et `GAMMA`. Nous ne représentons que les attributs de l'hypothèse qui sont pertinents pour l'opérateur de séquence : *histo* et *alias*. Ces deux attributs sont mis à jour à chaque franchissement d'une transition en attente d'un événement.

```
<DETECT>
 <EVENTS>
    ALPHA : PACKET { ... } A ;
    BETA  : PACKET { ... } B ;
    GAMMA : PACKET { ... } C ;
 </EVENTS>

 <ENCHAIN>
   (A ; B ; C)
 </ENCHAIN>
    ...
</DETECT>
```

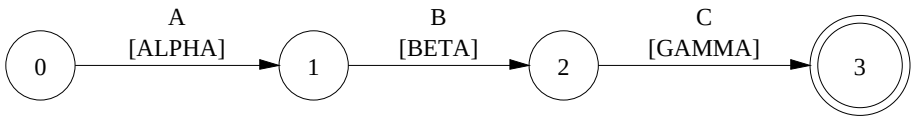

FIG. 3.2 – Automate équivalent à une séquence

Le flux des événements entrants considéré est : [a1,b1,a2,b2,c1].

Evénement	Transition	Hypothèse(s) créée(s)
		etat 0 : h1 = (histo=[[]], alias={ })
a1	0→1	etat 1 : h2 = (histo=[[a1]], alias={A=a1})
b1	1→2	etat 2 : h3 = (histo=[[a1,b1]], alias={A=a1,B=b1})
a2	0→1	etat 1 : h4 = (histo=[[a2]], alias={A=a2})
b2	1→2	etat 2 : h5 = (histo=[[a1,b2]], alias={A=a1,B=b2}) h6 = (histo=[[a2,b2]], alias={A=a2,B=b2})
c1	2→3	etat 3 : h7* = (histo=[[a1,b1,c1]], alias={A=a1,B=b1,C=c1}) h8* = (histo=[[a1,b2,c1]], alias={A=a1,B=b2,C=c1}) h9* = (histo=[[a2,b2,c1]], alias={A=a2,B=b2,C=c1})

TAB. 3.1 – Exemple de reconnaissance d'une séquence

Le marquage initial de l'automate a créé une hypothèse vide *h1* dans l'état 0 (en attente de l'événement `A`). L'apparition de l'événement *a1* active la transition en attente de l'événement `A`. Une nouvelle hypothèse *h2* (dérivée de *h1*) est alors créée dans l'état 1. L'événement *b1* provoque la création dans l'état 2 d'une nouvelle hypothèse *h3* (dérivée de *h2*). L'événement *a2* provoque la création dans l'état 1 d'une seconde hypothèse *h4* (dérivée de *h1*). L'événement *b2* provoque la

création dans l'état 2 des deux hypothèses *h5* et *h6* (dérivées respectivement de *h2* et *h4*). L'événement *c1* provoque la création dans l'état 3 (état d'acceptation) des trois hypothèses *h7*, *h8* et *h9*.

La notation '*' dans le tableau indique que chacune de ces hypothèses doit faire l'objet d'une alerte, puis doit être supprimée. On note que les hypothèses intermédiaires *h1* à *h6* restent dans leurs états respectifs afin de conserver tous les débuts de reconnaissance déjà calculés.

3.3.3 Opérateur de disjonction "One_among"

Si l'attaque dont on souhaite écrire la signature admet plusieurs variantes légères dans l'une de ses étapes, on peut utiliser l'opérateur `One_among` pour factoriser la reconnaissance de plusieurs signatures en une seule (cf paragraphe 2.3.1.2, p. 48).

Dans nos automates, on simule l'alternative en permettant aux hypothèses d'emprunter plusieurs chemins possibles dans l'automate. A cet effet, nous avons créé deux nouveaux types d'état : `MULTI` et `MONO`, qui matérialisent respectivement le début et la fin du sous-automate[4] correspondant à un opérateur `One_among`.

L'état transitoire[5] de type `MULTI` constitue un embranchement vers chacun des sous-automates alternatifs. Lorsqu'une hypothèse arrive dans cet état, elle est copiée en plusieurs exemplaires dont chacun est délivré, grâce à une ε-transition, à l'état de début du sous-automate correspondant.

Chacune des hypothèses va alors évoluer en parallèle des hypothèses situées dans les autres branches jusqu'à un état commun de type `MONO`.

L'état transitoire de type `MONO` ne comporte qu'une seule ε-transition en sortie. Lorsqu'une hypothèse arrive dans cet état, elle est automatiquement délivrée à l'unique état suivant (ce qui permet de resynchroniser les hypothèses sur un chemin unique).

On note que l'opérateur `One_among` est commutatif. En effet, l'automate résultant remplit la même fonction de reconnaissance quel que soit l'ordre de déclaration des sous-scénarios au sein de l'opérateur. Cela tient au fait que les sous-automates associés à chaque sous-scénario jouent un rôle symétrique (hypothèses progressant en parallèle).

Exemple

La figure 3.3 représente un exemple de scénario (constitué de la mise en sé-

[4] dans l'exemple de la figure 3.3, ce sont les états numérotés 1 et 4.

[5] c'est-à-dire qui ne stocke pas d'hypothèses

quence d'un événement et d'un opérateur `One_among`) ainsi que l'automate équivalent.

Les conventions graphiques utilisées dans cette figure sont les suivantes :

- un triangle simple en traits pointillés représente un état transitoire de type `MULTI` qui symbolise le début d'un opérateur `One_among` ;
- une flèche en traits pointillés représente une ε-transition qui délivre (sans conditions) une hypothèse envoyée par un état source à un état destination ;
- un triangle double en traits pointillés représente un état transitoire de type `MONO` qui symbolise la fin d'un opérateur `One_among`.

```
<DETECT>
 <EVENTS>
    ALPHA : SNARE { ... } A ;
    BETA  : SNARE { ... } B ;
    GAMMA : SNARE { ... } C ;
 </EVENTS>

 <ENCHAIN>
   (A ; One_among{B , C})
 </ENCHAIN>
    ...
</DETECT>
```

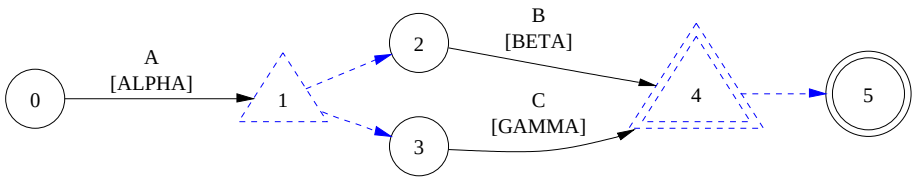

FIG. 3.3 – Automate équivalent à un scénario contenant un `One_among`

Le tableau 3.2 représente un exemple de la reconnaissance du scénario précédent (figure 3.3). Nous notons ai, bi et ci les événements concrets typés respectivement `ALPHA`, `BETA` et `GAMMA`. Nous ne représentons que les attributs de l'hypothèse qui sont pertinents pour l'opérateur `One_among` : *histo* et *alias*. Ces deux attributs sont mis à jour à chaque franchissement d'une transition en attente d'un événement.

Le flux des événements entrants considéré est : [a1,b1,c1].

Evénément	Transition	Hypothèse(s) créée(s)
		etat 0 : h1 = (histo=[[ ]], alias={ })
a1	0→1	etat 2 : h2 = (histo=[[a1]], alias={A=a1})
		etat 3 : h3 = (histo=[[a1]], alias={A=a1})
b1	2→4	etat 5 : h4* = (histo=[[a1,b1]], alias={A=a1,B=b1})
c1	3→4	etat 5 : h5* = (histo=[[a1,c1]], alias={A=a1,C=c1})

TAB. 3.2 – Exemple de reconnaissance d'un `One_among`

Le marquage initial de l'automate a créé une hypothèse vide *h1* dans l'état 0 (en attente de l'événement `A`). L'apparition de l'événement *a1* active la transition en attente de l'événement `A`. Une hypothèse (dérivée de *h1*) est envoyée à l'état 1 (de type `MULTI`). Elle est immédiatement dupliquée et donne lieu à la création des hypothèses *h2* (dans l'état 2) et *h3* (dans l'état 3). L'événement *b1* active la transition en attente de l'événement `B`. Une hypothèse (dérivée de *h2*) est envoyée à l'état 4 (de type `MONO`). Elle est immédiatement transmise à l'état 5 ce qui donne lieu à la création de l'hypothèse *h4*. L'événement *c1* active la transition en attente de l'événement `C`. Une hypothèse (dérivée de *h3*) est envoyée à l'état 4 (de type `MONO`). Elle est immédiatement transmise à l'état 5 ce qui donne lieu à la création de l'hypothèse *h5*.

L'état 5 est aussi l'état d'acceptation : les hypothèses *h4* et *h5* sont utilisées pour construire des alertes puis sont supprimées.

3.3.4 Opérateur de conjonction "`Non_ordered`"

L'opérateur `Non_ordered` représente la conjonction de la reconnaissance de plusieurs sous-scénarios dans un ordre quelconque et sans événements partagés entre les sous-scénarios (cf paragraphe 2.3.1.2, p. 49). La reconnaissance de ces sous-scénarios s'effectue donc en parallèle et les événements qui les composent peuvent être entrelacés dans le flux des événements entrants.

Dans nos automates, on simule le parallélisme par un éclatement d'une hypothèse en plusieurs hypothèses partielles qui vont progresser dans des sous-automates différents en parallèle et qui seront refusionnées dès que chacun des sous-automates aura été reconnu. A cet effet, nous avons créé deux nouveaux types d'état : `DISTRIB` et `MERGE`, qui matérialisent respectivement le début et la fin du sous-automate[6] correspondant à un opérateur `Non_ordered`.

L'état transitoire de type `DISTRIB` constitue un embranchement vers chacun des sous-automates à reconnaître en parallèle. Lorsqu'une hypothèse arrive dans

[6]dans l'exemple de la figure 3.4, ce sont les états numérotés 0 et 3.

cet état, on lui attribue un numéro de génération unique (ajouté à son attribut *sync*[7]) qui sera transmis à toutes les hypothèses dérivées. Un nouvel historique vide est ajouté à son attribut *histo* (qui représentera un historique local des événements qui font progresser l'hypothèse). L'hypothèse ainsi modifiée est ensuite copiée en plusieurs exemplaires (appelées hypothèses partielles) dont chacun est délivré, grâce à une ε-transition, à l'état de début du sous-automate correspondant.

Chacune des hypothèses va alors évoluer en parallèle des hypothèses situées dans les autres branches jusqu'à un état commun de type `MERGE`.

L'état de type `MERGE` constitue le point de synchronisation où toutes les hypothèses partielles (de même génération) devront être refusionnées. Lorqu'une hypothèse partielle arrive dans cet état par l'un des branches, elle est conservée dans une liste dédiée à cette branche (via l'attribut *courant*). Si chacune des autres listes comporte au moins une hypothèse partielle de la même génération, on a potentiellement reconnu tous les sous-automates. On va alors essayer de fusionner ces hypothèses partielles afin de reconstituer une hypothèse «normale». On énumère donc tous les n-uplets d'hypothèses partielles possibles (1 hypothèse par liste) et on tente de fusionner les valeurs contenues dans les différents attributs. Pour chaque n-uplet, les historiques locaux (qui ne doivent pas contenir d'événements communs) sont fusionnés en un seul, trié par ordre de temps croissant. L'historique obtenu deviendra l'historique local de la nouvelle hypothèse (qui comporte maintenant un niveau d'historique de moins que les hypothèses partielles). L'hypothèse issue de ce processus de fusion est ensuite envoyée à l'état suivant.

On note que l'opérateur `Non_ordered` est commutatif (pour les mêmes raisons que celles évoquées précédemment à propos de l'opérateur `One_among`).

Exemple

La figure 3.4 représente un exemple de scénario (constitué de la mise en séquence d'un opérateur `Non_ordered` et d'un événement) ainsi que l'automate équivalent.

Les conventions graphiques utilisées dans cette figure sont les suivantes :

- un parallélogramme simple en traits pointillés représente un état transitoire de type `DISTRIB` qui symbolise le début d'un opérateur `Non_ordered` ;
- un parallélogramme double en traits pointillés représente un état de type `MERGE` qui symbolise la fin d'un opérateur `Non_ordered`.

[7] *sync* est de type liste ce qui permet de gérer plusieurs niveaux d'imbrication d'opérateurs `Non_ordered`.

```
<DETECT>
 <EVENTS>
    ALPHA : BSM { ... } A ;
    BETA  : BSM { ... } B ;
    GAMMA : BSM { ... } C ;
 </EVENTS>

 <ENCHAIN>
   (Non_ordered{A , B} ; C)
 </ENCHAIN>
    ...
</DETECT>
```

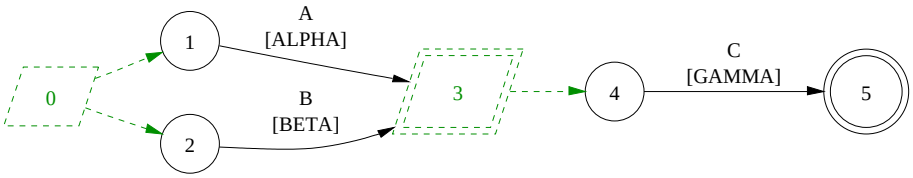

FIG. 3.4 – Automate équivalent à un scénario contenant un `Non_ordered`

Le tableau 3.3 représente un exemple de la reconnaissance du scénario précédent (figure 3.4). Nous notons ai, bi et ci les événements concrets typés respectivement `ALPHA`, `BETA` et `GAMMA`. Nous ne représentons que les attributs de l'hypothèse qui sont pertinents pour l'opérateur `Non_ordered` : *histo*, *alias* et *sync*. L'attribut *sync* est modifié lorsqu'une hypothèse entre dans des états de type `DISTRIB` ou `MERGE`.

Le flux des événements entrants considéré est : [a1,b1,a2,c1].

Le marquage initial de l'automate a envoyé une hypothèse vide dans l'état 0 (de type `DISTRIB`). Cette hypothèse s'est vue attribuer un nouveau numéro de génération (ajouté à l'attribut *sync*) ainsi qu'un niveau supplémentaire d'historique (vide). Elle est ensuite dupliquée ce qui crée deux hypothèses partielles vides en attente d'événements (*h1* dans l'état 1 et *h2* dans l'état 2). L'événement *a1* provoque la création d'une nouvelle hypothèse *h3* (dérivée de *h1*) dans l'état 3 (de type `MERGE`), qui est stockée dans la liste des événements provenant de l'état 1. Comme il n'y a pas encore d'événements provenant de l'état 2, le processus de fusion des hypothèses partielles n'est pas lancé. L'événement *b1* provoque la création d'une nouvelle hypothèse *h4* (dérivée de *h2*) dans l'état 3, qui est stockée dans la liste des événements provenant de l'état 2. On peut maintenant réaliser

la fusion des hypothèses *h3* et *h4* (de même numéro de génération). Une nouvelle hypothèse *h5* est créée et envoyée dans l'état 4. L'événement *a2* provoque la création d'une nouvelle hypothèse *h6* (dérivée de *h1*) dans l'état 3. On réalise la fusion des hypothèses *h4* et *h6* (de même numéro de génération). Une nouvelle hypothèse *h7* est alors créée et envoyée dans l'état 4. L'événement *c1* provoque la création des hypothèses *h8* et *h9* (dérivées resp. des hypothèses *h5* et *h7*) dans l'état 5, qui est l'état d'acceptation. Les hypothèses *h8* et *h9* sont utilisées pour construire des alertes puis sont supprimées.

Evénément	Transition	Hypothèse(s) créée(s)
		etat 1 : h1 = (histo=[[ ],[ ]], alias={ }, sync=[0,1]) etat 2 : h2 = (histo=[[ ],[ ]], alias={ }, sync=[0,1])
a1	1→3	etat 3 : h3 = (histo=[[ ],[a1]], alias={A=a1}, sync=[0,1])
b1	2→3	etat 3 : h4 = (histo=[[ ],[b1]], alias={B=b1}, sync=[0,1]) etat 4 : h5 = (histo=[[a1,b1]], alias={A=a1,B=b1}, sync=[0])
a2	1→3	etat 3 : h6 = (histo=[[ ],[a2]], alias={A=a2}, sync=[0,1]) etat 4 : h7 = (histo=[[b1,a2]], alias={B=b1,A=a2}, sync=[0])
c1	4→5	etat 5 : h8* = (histo=[[a1,b1,c1]], alias={A=a1,B=b1,C=c1}, sync=[0]) h9* = (histo=[[b1,a2,c1]], alias={B=b1,A=a2,C=c1}, sync=[0])

TAB. 3.3 – Exemple de reconnaissance d'un `Non_ordered`

3.3.5 Opérateur d'exclusion "`Without`"

L'opérateur `Without` permet d'exprimer la reconnaissance d'un premier scénario (dit positif) durant laquelle un second scénario (dit négatif) ne doit pas être reconnu sous peine d'invalider les hypothèses liées à la reconnaissance du premier (cf paragraphe 2.3.1.2, p. 50).

Dans nos automates, nous simulons ce comportement en faisant évoluer en parallèle les sous-automates correspondant à chacun des deux scénarios (positif et négatif) et en introduisant un point de synchronisation qui détermine lequel est reconnu le premier. A cet effet, nous avons créé deux nouveaux types d'état : `WITHOUT` et `SYNC`, qui matérialisent respectivement le début et la fin du sous-automate[8] correspondant à un opérateur `Without`.

L'état transitoire de type `WITHOUT` marque le début du sous-automate positif. Lorsqu'une hypothèse arrive dans cet état, on lui attribue un numéro de généra-

[8]dans l'exemple de la figure 3.5, ce sont les états numérotés 0 et 10 (ou 2 et 6).

tion unique (ajouté à son attribut *with*[9]) qui sera transmis à toutes les hypothèses dérivées. Cette hypothèse modifiée est alors transmise à l'état suivant. Selon le scénario positif originel, elle donnera lieu à la création d'une ou plusieurs[10] hypothèses dans des états de type `NORMAL`, qui constituent autant d'états «de départ» pour la reconnaissance du sous-automate positif.

Les transitions issues de l'un de ces états de départ comportent une liste (attribut *inhib*) des états initiaux des sous-automates négatifs à activer. Dès qu'un événement active l'une de ces transitions et fait progresser une hypothèse, on réalise un marquage initial du (ou des) sous-automate(s) négatif(s) correspondant(s). C'est cette hypothèse, et non pas une hypothèse vide, qui est utilisée pour le marquage initial. Cela signifie que les hypothèses d'un sous-automate négatif comportent le même numéro de génération que l'hypothèse qui a déclenché le sous-automate négatif.

Les hypothèses des sous-automates positifs et négatifs (de même génération) progressent en parallèle jusqu'à un état commun de type `SYNC`. Lorsqu'une hypothèse arrive dans un état de ce type, il y a deux cas de figure. Si l'hypothèse arrive par la branche positive, alors le sous-automate positif a été entièrement reconnu sans être invalidé. Dans ce cas, on modifie l'hypothèse en lui retirant le dernier élément de son attribut *with*, puis on la délivre à l'état suivant grâce à une ε-transition. Par contre, si l'hypothèse arrive par la branche négative, alors le sous-automate négatif a été entièrement reconnu. Il faut donc invalider la reconnaissance du sous-automate positif correspondant ; cela signifie détruire toutes les hypothèses ayant le même numéro de génération que cette hypothèse, qu'elles soient présentes dans le sous-automate positif ou dans le sous-automate négatif.

Exemple

La figure 3.5 représente un exemple de scénario (constitué de l'imbrication de deux opérateurs `Without`) ainsi que l'automate équivalent.

Les conventions graphiques utilisées dans cette figure sont les suivantes :

- un hexagone simple en traits pointillés représente un état transitoire de type `WITHOUT` qui symbolise le début d'un opérateur `Without` ;
- une flèche en trait plein (annotée en dessous du mot clé 'inhib' et d'une liste de numéros d'états) représente une transition déclenchable sur occurence d'un événement qui doit également activer un ou plusieurs sous-automates négatifs ;
- un hexagone double en traits pointillés représente un état transitoire de type `SYNC` qui symbolise la fin d'un opérateur `Without`.

[9] *with* est de type liste ce qui permet de gérer plusieurs niveaux d'imbrication d'opérateurs `Without`.

[10] dans l'exemple de la figure 3.5, ce sont les états 3 et 7.

Evénément	Transition	Hypothèse(s) créée(s)
		etat 3 : h1 = (histo=[[ ],[ ]], alias={ }, sync=[0,1], with=[0,1,2]) etat 7 : h2 = (histo=[[ ],[ ]], alias={ }, sync=[0,1],with=[0,1])
e1	3→4	etat 4 : h3 = (histo=[[ ],[e1]], alias={E=e1}, sync=[0,1]) etat 5 : h4 = (histo=[[ ]], alias={ }, sync=[0], with=[0,2]) etat 9 : h5 = (histo=[[ ]], alias={ }, sync=[0], with=[0,1,2])
t1	7→8	etat 8 : h6 = (histo=[[ ],[t1]], alias={T=t1}, sync=[0,1], with=[0,1]) etat 9 : h7 = (histo=[[ ]], alias={ }, sync=[0], with=[0,1])
f1	4→6	etat 8 : h8 = (histo=[[ ],[e1,f1]], alias={E=e1,F=f1}, sync=[0,1], with=[0,1]) etat 11 : h9* = (histo=[[e1,t1,f1]], alias={E=e1,F=f1,T=t1}, sync=[0], with=[0])

TAB. 3.4 – Exemple de reconnaissance d'un `Without`

Le tableau 3.4 représente un exemple de la reconnaissance du scénario précédent (figure 3.5). Nous notons e*i*, f*i*, a*i*, t*i* et b*i* les événements concrets typés respectivement `RPCINFO`, `SHOWMOUNT`, `FINGER`, `MOUNT`, et `FAPPEND`. Nous représentons les attributs de l'hypothèse qui sont pertinents pour l'opérateur `Without` (*histo*, *alias* et *with*) ainsi que l'attribut *sync* car l'exemple contient un opérateur `Non_ordered`. L'attribut *with* est modifié lorsqu'une hypothèse entre dans des états de type `WITHOUT` ou `SYNC`.

Le flux des événements entrants considéré est : [e1,t1,f1].

Le marquage initial de l'automate envoie une hypothèse vide dans l'état 0 (de type `WITHOUT`). Cette hypothèse se voit attribuer un nouveau numéro de génération (ajouté à l'attribut *with*). Elle est ensuite transmise à l'état suivant. Par le jeu des opérateurs imbriqués, deux hypothèses initiales *h1* et *h2* sont créées respectivement dans les états 3 et 7. On note que l'attribut *with* de *h1* comporte un numéro de génération supplémentaire par rapport à celui de *h2* car l'état 3 est sous la portée de deux opérateurs `Without`.

L'événement *e1* provoque la création d'une nouvelle hypothèse *h3* (dérivée de *h1*) dans l'état 4. Il déclenche également le marquage initial des sous-automates négatifs associés à la transition (attribut *inhib*), ce qui crée les hypothèses *h4* et *h5* (dérivées de *h3*) dans les états 5 et 9.

L'événement *t1* provoque la création d'une nouvelle hypothèse *h6* (dérivée de *h2*) dans l'état 8 (de type `MERGE`). Aucune fusion n'est encore possible. Il déclenche également le marquage initial du sous-automate négatif associé à la transition, ce qui crée une hypothèse *h7* dans l'état 9.

```
<DETECT>
  <EVENTS>
    RPCINFO   : PACKET { ... } E ;
    SHOWMOUNT : PACKET { ... } F ;
    FINGER    : PACKET { ... } A ;
    MOUNT     : PACKET { ... } T ;
    FAPPEND   : SNARE  { ... } B ;
  </EVENTS>

  <ENCHAIN>
      [ Non_ordered{ [(E;F)] Without{A} , T} ] Without {B}
  </ENCHAIN>
    ...
</DETECT>
```


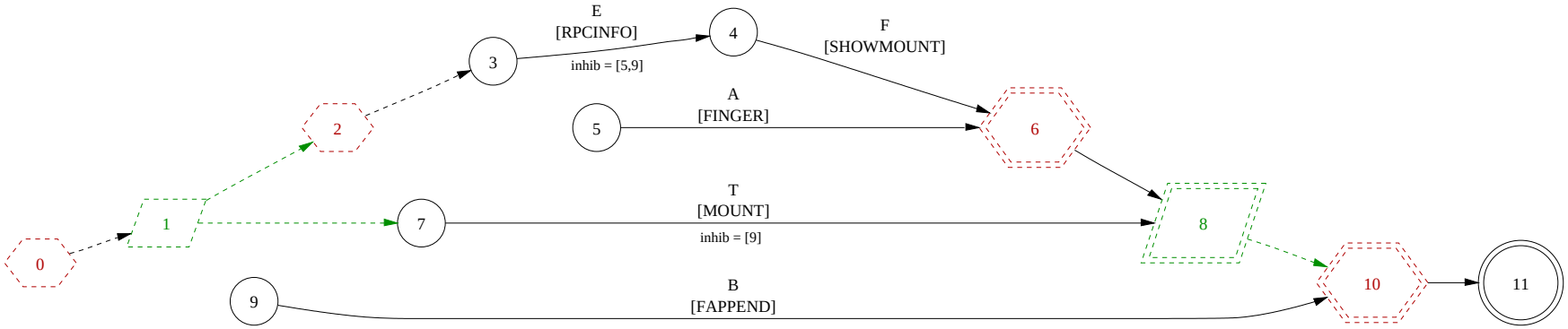

FIG. 3.5 – Automate équivalent à une imbrication de Without

L'événement *f1* provoque la création d'une nouvelle hypothèse *h8* (dérivée de *h3*) dans l'état 8 (après avoir traversé l'état 6 de type `SYNC`). On peut maintenant réaliser une fusion entre les hypothèses partielles *h6* et *h8*. L'hypothèse résultante est délivrée à l'état 10 (de type *SYNC*). Comme elle provient d'un sous-automate positif, elle est modifiée (perte de son numéro de génération) puis délivrée à l'état 11 (de type *FINAL*) qui en fait l'hypothèse *h9*. L'hypothèse *h9* est utilisée pour construire une alerte puis supprimée.

3.3.6 Contrainte temporelle "Timeout"

Lorsque l'on déclare une contrainte de type `Timeout` dans la section `<ENCHAIN>`, on donne une «durée de vie» maximale aux hypothèses qui évoluent dans l'automate de reconnaissance (sauf pour les hypothèses initiales introduites à l'initialisation de l'automate qui doivent rester en attente d'un événement).

Une hypothèse initiale qui franchit sa première transition reçoit une date de péremption (calculée en ajoutant la durée de vie à l'heure de l'événement courant) qui est conservée dans son attribut *timeout*. A chaque itération de l'algorithme de détection, la première phase consiste à rechercher dans l'automate les hypothèses périmées et les détruire.

Ce type de contrainte définit une coupure dans le processus de détection. Elle est peut être utilisée dans deux cas de figure. Premièrement, certaines attaques doivent se dérouler dans un intervalle de temps maximal. On peut donc abandonner une reconnaissance partielle (représentée par une hypothèse) commencée depuis plus longtemps que cette durée maximale. Deuxièmement, cette fonctionnalité est utile pour lutter contre le risque d'explosion combinatoire inhérent à notre modèle (étant donné que l'algorithme ajoute régulièrement des nouvelles hypothèses mais en supprime rarement). Cela signifie qu'on décide alors explicitement de ne considérer que les attaques qui sont entièrement reconnues dans une fenêtre temporelle fixe. On accepte donc le risque d'avoir des faux-négatifs.

3.3.7 Contrainte temporelle "MinDelay"

Cet opérateur définit une contrainte de délai minimum à respecter entre deux événements particuliers. Une hypothèse (ayant franchi la première transition) ne peut pas franchir la transition correspondant au second événement tant qu'un certain délai (en secondes) ne s'est pas encore écoulé.

Soient E_i et E_j deux instances d'événements, d un délai en secondes, alors la contrainte

$$\texttt{MinDelay(}E_i\texttt{,}E_j\texttt{) == }d$$

possède la propriété suivante :

$$|t(E_i) - t(E_j)| \leq d$$

Exemple : `MinDelay(E1,E2) == 2`

Comme on ne sait pas forcément l'ordre dans lequel apparaitront les événements considérés (ex : les événements sont déclarés dans des branches distinctes d'un opérateur `Non_ordered`), la contrainte est séparée en deux contraintes symétriques.

Chaque événement se voit réserver une variable (ex : `T_E1` pour `E1` et `T_E2` pour `E2`) destinée à contenir l'horodatage de l'événement correspondant. Ensuite, deux gardes symétriques sont générées. La première est portée par la transition associée à `E1` : `Unif2(T_E1,MINDELAY,2,T_E2)`. La seconde est portée par la transition associée à `E2` : `Unif2(T_E2,MINDELAY,2,T_E1)`.

Lorsqu'une hypothèse s'apprête à franchir la première des deux transitions (ex : `E2`), la contrainte ne peut pas encore être évaluée car la variable `T_E1` n'existe pas encore dans son environnement. L'horodatage de l'événement courant est alors mis dans la variable `T_E2` laquelle est ajoutée à l'environnement porté par l'hypothèse (attribut *env*).

Lorsque cette hypothèse tente de franchir la transition associée au second événément (`E1`), la contrainte peut être évaluée. L'horodatage courant est mis dans la variable `T_E1`. On peut alors vérifier que le délai entre `T_E2` et `T_E1` n'est pas dépassé, auquel cas la garde est satisfaite et la transition peut être franchie.

Dans le cas particulier où les deux événements contraints sont situés dans deux branches différentes d'un sous-automate équivalent à un opérateur `Non_ordered`, ce sont deux hypothèses partielles distinctes qui tenteront d'évaluer les contraintes. Il faut donc attendre l'état de resynchronisation pour les évaluer lors du processus de fusion des hypothèses.

Ce type de contrainte définit également une coupure dans le processus de détection car on supprime des reconnaissances d'événement qui pouvaient potentiellement faire progresser des hypothèses et donner lieu à une alerte.

3.3.8 Contrainte temporelle "MaxDelay"

Cet opérateur définit une contrainte de délai maximum à respecter entre deux événements particuliers. Une hypothèse qui a franchi la transition associée au premier ne peut franchir la transition associée au second seulement que si un certain délai (en secondes) n'est pas encore dépassé.

Soient E_i et E_j deux instances d'événements, d un délai en secondes, alors la contrainte

`MaxDelay(`E_i`,`E_j`) ==` d

possède la propriété suivante :

$$|t(E_i) - t(E_j)| \geq d$$

Le principe utilisé pour permettre la vérification de cette contrainte est exactement le même que pour `MinDelay`. On utilise de la même façon des variables d'environnement et des gardes de type `Unif2` (mais avec l'opérateur `MAXDELAY`).

Une hypothèse qui échoue au test de MaxDelay doit être éliminée car elle ne pourra jamais franchir la transition étant donné que des événements successifs ont un horodatage croissant.

Ce type de contrainte définit une coupure dans le processus de détection car, comme pour la contrainte `Timeout`, on donne une durée de vie maximale aux hypothèses (mais qui ne s'applique qu'après franchissement d'une transition particulière).

3.4 Sémantique de la corrélation contextuelle

Dans le langage, la section `<CONTEXT>` permet d'exprimer des contraintes dites contextuelles (cf 2.3.1.3, p. 53). Elles peuvent être de type intra-événement (c'est-à-dire qu'elles portent uniquement sur les valeurs contenues dans un événement) ou inter-événements (c'est-à-dire qu'elles lient entre elles plusieurs instances d'événements provenant de la même attaque).

Dans les sections suivantes, nous donnons la sémantique opérationnelle associée à ces contraintes.

3.4.1 Contraintes intra-événement

Une contrainte intra-événement représente une étape de filtrage supplémentaire sur les événements. Elle correspond à une condition logique que doit vérifier un événement particulier pour pouvoir activer la transition à laquelle il est associé.

Dans notre modèle à base d'automates, nous traduisons cette contrainte en une garde qui sera ajoutée à l'attribut *gardes* de la transition associée à l'événement. Selon que la contrainte porte sur un ou deux champs de l'événement, la garde associée sera de type `Intra` ou `Intra2`.

Par exemple, si l'on souhaite qu'un événement nommé `E0` (défini dans la section `<EVENTS>` comme un événement de type `LOGIN` issu de l'audit système `SNARE`) corresponde à une activité de l'administrateur, on peut écrire la contrainte suivante :

```
<CONTEXT>
  E0/snare/user/uid == "root(0)"
</CONTEXT>
```

Nous définissons alors une garde `Intra(E0,"snare/user/uid",EQ,"root(0)")` qui sera portée par la transition associée à EO.

3.4.2 Contraintes inter-événements

Une contrainte inter-événements représente une corrélation logique entre plusieurs événements. Cela signifie qu'une hypothèse devra vérifier des conditions au franchissement de plusieurs transitions dans l'automate. Ce type de contrainte se traduit par l'ajout de gardes sur les transitions concernées et par l'utilisation des variables d'environnement liées aux hypothèses pour transporter un contexte.

On distingue deux cas de figure selon que l'opérateur[11] utilisé dans les contraintes est l'égalité ou bien une inégalité.

Egalité entre les champs de plusieurs événements

Si l'on déclare une contrainte reliant plusieurs événements par une relation d'égalité sur l'un de leurs champs, une garde (référençant une variable d'environnement partagée) est créée pour chacune des transitions concernées.

Exemple :

```
<CONTEXT>
 E0/snare/user/uid == E1/snare/user/euid == E2/snare/user/gid
</CONTEXT>
```

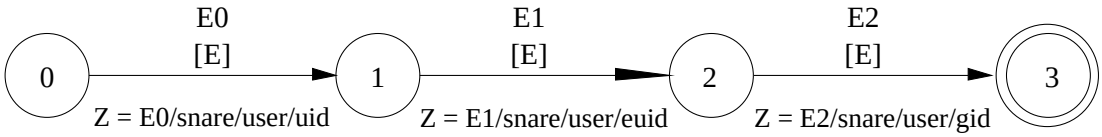

FIG. 3.6 – Contrainte d'égalité entre événements

Dans l'exemple de la figure 3.6, la variable partagée est nommée arbitrairement `Z`. Elle est utilisée pour la définition des gardes portées par les transitions associées aux événements `E0`, `E1` et `E2`. Ces gardes sont déclarées respectivement `Unif(E0,EQ,Z,snare/user/uid)`, `Unif(E1,EQ,Z,snare/user/euid)` et `Unif(E2,EQ,Z,snare/user/gid)`. Dans la figure, nous les avons représentées de manière simplifiée sous chacune des transitions.

Lorsqu'une hypothèse est candidate au franchissement d'une transition comportant une garde de ce type, deux cas de figure se présentent.

Si la variable n'existe pas encore dans l'environnement de l'hypothèse, alors on crée la variable (par effet de bord) et on lui affecte la valeur du champ concerné de

[11] Dans la version actuelle de la grammaire, il n'est pas possible d'exprimer des contraintes de type arithmétique (addition, soustraction, multiplication, division ...).

l'événement. On considère que la garde est vérifiée. La variable d'environnement sera désormais transportée avec l'hypothèse (attribut *env*) dans la mesure où celle-ci vérifie l'ensemble des gardes portées par cette transition.

Si la variable existe déjà dans l'environnement de l'hypothèse, elle contient la valeur commune aux différents champs liés par la contrainte. On peut alors comparer la valeur du champ de l'événement courant avec le contenu de la variable. S'il y a égalité, la garde est vérifiée et l'hypothèse peut potentiellement franchir la transition (si toutes les gardes associées à la transition sont vérifiées). Sinon, l'événement courant ne respectait pas la contrainte ; il ne peut donc pas faire progresser l'hypothèse courante.

La syntaxe du langage permet également une variante pour la déclaration de contraintes de ce type (où chaque champ est lié à une variable commune explicite). Exemple :

```
<CONTEXT>
  X := E0/packet/ether/ip/tcp/sport
  E1/packet/ether/ip/tcp/dport == X
</CONTEXT>
```

Le principe reste exactement le même, mais c'est le nom de cette variable qui est utilisé dans les gardes (au lieu d'un nom arbitraire).

Dans certains cas, les événements contraints sont situés dans des branches différentes d'un sous-automate équivalent à un opérateur `Non_ordered`. Il faut attendre l'état de resynchronisation pour vérifier (lors du processus de fusion des hypothèses) si les variables d'environnement de chacune des hypothèses partielles sont compatibles.

Inégalité entre les champs de deux événements

Si l'on déclare une contrainte reliant deux événements par une relation d'inégalité (différence, infériorité ou supériorité), une garde (référençant deux variables d'environnement partagées) est créée pour chacune des deux transitions associées. Ces deux gardes comportent des opérateurs symétriques[12] qui assurent que la condition sera évaluée quel que soit l'ordre d'apparition des deux événements. En effet, connaissant la signature temporelle, il est généralement possible de prévoir lequel des deux événements sera atteint en premier par une hypothèse (et donc stocker la première valeur rencontrée puis comparer avec la seconde). Toutefois, nous n'avons pas souhaité utiliser des actions sur les transitions ; c'est pourquoi nous avons choisi de permettre aux gardes de faire des effets de bord (affectation).

[12] Aux opérateurs '!=', '<', '>', '<='et '>=' correspondent respectivement les opérateurs '!=', '>=', '<=', '>'et '<'.

Exemple :

```
<CONTEXT>
  E0/packet/ether/ip/tcp/sport < E1/packet/ether/ip/tcp/dport
</CONTEXT>
```

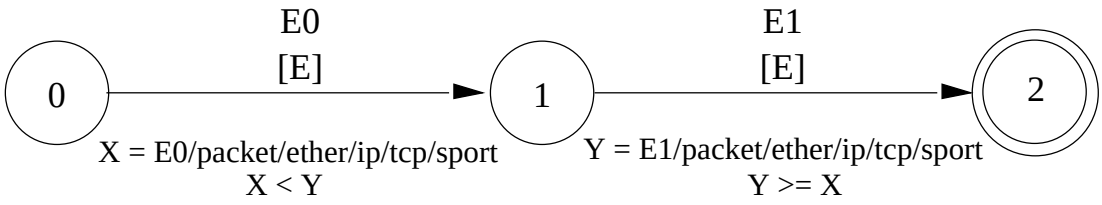

FIG. 3.7 – Contrainte d'inégalité entre événements

Dans l'exemple de la figure 3.7, deux variables partagées sont nommées arbitrairement `X` et `Y`. Elles sont utilisées pour la définition des gardes portées par les transitions associées aux événements `E0` et `E1`. La garde déclarée pour l'événement `E0` est `Unif3(E0,packet/ether/ip/tcp/sport,X,INF,Y)`. Celle définie pour `E1` est `Unif3(E1,packet/ether/ip/tcp/dport,Y,SUPEQ,X)`. Dans la figure, nous les avons représentées de manière simplifiée sous chaque transition.

Lorsqu'une hypothèse doit franchir une transition comportant une garde de ce type, la vérification se fait en deux étapes. La valeur du champ concerné de l'événement courant est affectée à la première variable d'environnement (par un effet de bord de la garde). Ensuite, on tente d'évaluer la relation entre cette variable et la seconde variable partagée.

Si la seconde variable n'existe pas encore dans l'environnement de l'hypothèse, la relation ne peut pas être évaluée : on considère que la garde est vérifiée et l'hypothèse qui franchit la transition transporte désormais la première variable dans son environnement. La relation pourra donc être testée ultérieurement (lors du franchissement de la transition associée à l'autre événement contraint).

Si la seconde variable existe déjà dans l'environnement, c'est le résultat booléen de la comparaison entre les deux variables qui décide si la garde est vérifiée et, par conséquent, si la transition peut éventuellement être franchie.

Dans certains cas, les événements contraints sont situés dans des branches différentes d'un sous-automate équivalent à un opérateur `Non_ordered` . Il faut attendre l'état de resynchronisation pour vérifier (lors du processus de fusion des hypothèses) si les variables d'environnement de chacune des hypothèses partielles sont compatibles.

Remarque :
L'approche utilisée pour la résolution des contraintes d'inégalités inter-événements peut conduire, dans certaines situations, à des hypothèses surnuméraires au sein de l'automate de reconnaissance (dont les contraintes ne pourront jamais être résolues). Par exemple, si un champ d'un événement (représenté par une variable `Y`) intervient dans deux contraintes d'inégalité différentes et que les relations résultantes sont `X<Y` et `Y<Z`, et si les variables `X` et `Z` sont instanciées mais que leurs valeurs ne respectent pas la relation de transitivité précédente, alors on peut déjà conclure qu'il n'y a pas de solution possible quel que soit `Y`. Or, notre modèle actuel de résolution n'élimine pas les hypothèses qui sont dans cette situation. Ce type d'hypothèses ne peut toutefois pas mener à des alertes erronées puisqu'elles n'atteindront jamais l'état final de l'automate.

Pour remédier à ce problème, il faudrait analyser toutes les contraintes inter-événements dans leur ensemble et inférer toutes les relations implicites qui peuvent exister (et en déduire de nouvelles gardes qui seront attachées à des transitions).

3.5 Principes de l'algorithme abstrait utilisé pour la détection

Nous proposons un algorithme pour la détection d'une attaque qui s'appuie sur un automate à états finis pour traiter un flux d'événements (aussi appelé Log). Nous présentons ici de manière informelle les différentes étapes de l'algorithme par une description simplifiée du graphe d'appel.

Le détail de l'algorithme se trouve en annexe F.

La fonction principale `Detection()` constitue le point d'entrée de l'algorithme. Elle construit l'automate de reconnaissance (`constructionAuto`), l'initialise (`initialisationAuto`) puis lance l'analyse du Log (`analyseLog`).

La procédure `analyseLog()` fonctionne de façon incrémentale en délivrant les événements un par un à l'automate. Elle permet de traiter de manière identique le cas de la détection *on-line* et celui de la détection *off-line*. Si l'analyse se fait *on-line*, la fonction `existeElementSuivant()` renvoie toujours vrai mais la fonction `renvoieElementSuivant()` devient une lecture bloquante tant que la file d'attente représentant le flux d'événements est vide. Si l'analyse se fait *off-line*, la fonction `existeElementSuivant()` ne renvoie faux que lorsque tous les événements concrets du Log ont été consommés.

La fonction `filtrage()` décide pour chaque événement du Log s'il est intéressant pour l'attaque courante et renvoie, le cas échéant, un événement synthétique (type abstrait `Evenement`, cf 3.2.1) qui est désormais typé, horodaté et dont les

informations intrinsèques sont modélisées sous la forme d'un ensemble de paires champ-valeur. Cet événement synthétique est alors présenté à l'automate par un appel à `utiliseEvt`.

La procédure `utiliseEvt()` correspond à un pas élémentaire d'éxécution de l'algorithme puisqu'elle permet à l'automate de traiter un événement. Elle se déroule en quatre phases :

- la phase de suppression (`destructionTimeout`) des hypothèses dont la durée de vie (au vu de l'horodatage de l'événement courant) est dépassée ;
- la phase de prise en compte de l'événement par l'automate (`distribueEvt`) qui constitue le cœur de l'algorithme ;
- la phase de suppression (`supprimeHypotheses`) des hypothèses marquées «à détruire» par la seconde phase ;
- la phase d'ajout (`majEtats`) à l'automate des hypothèses créées au pas d'exécution courant pour qu'elles soient prises en compte au pas suivant.

La procédure `distribueEvt()` propose l'événement courant à chacune des transitions de l'automate qui sont compatibles (même type d'événement attendu) et qui comportent un état actif en amont. Si l'événement passe les gardes intra-événement, alors on traite chaque hypothèse. Pour chaque hypothèse, on teste les gardes inter-événements. Si l'hypothèse a passé tous les tests, alors on copie ses informations qui sont ensuite modifiées pour tenir compte de la reconnaissance de l'événement courant et envoyées à l'état aval de la transition par un appel à `ajoutHypothese()`.

La procédure récursive `ajoutHypothese()` détermine les actions effectuées sur l'hypothèse selon le type de l'état d'arrivée. Si l'état est de type `NORMAL`, l'hypothèse est stockée et la récursion s'arrête. Si l'état est de type MERGE, l'hypothèse est également conservée et une fusion avec d'autres hypothèses est tentée. Les états de type `DISTRIB, MERGE, MULTI, MONO, WITHOUT et SYNC` appellent récursivement la fonction pour propager des hypothèses.

Finalement, lorsque l'état atteint par l'hypothèse est l'état d'acceptation de l'automate (type `FINAL`), il y a reconnaissance complète de la signature et émission d'une alerte (`envoiAlerte`).

3.6 Problème de l'explosion combinatoire

Nous avons fait le choix de rechercher de manière **exhaustive** les occurences d'une signature dans un flux d'événements. Cela pose le problème du nombre

d'hypothèses présentes au sein de l'automate puisque, par défaut, l'algorithme garde en mémoire toutes les hypothèses intermédiaires.

On note que ce phénomène est accentué par la sémantique associée aux opérateurs `Non_ordered` et `One_among` (une hypothèse donnant lieu à n hypothèses).

Même si, dans la pratique, les scénarios réels comporteront un nombre réduit d'événements, nous disposons de deux solutions dans le langage pour lutter contre ce problème d'explosion combinatoire.

Premièrement, les contraintes contextuelles (de type intra-événement et inter-événements) et les contraintes temporelles (de type `MinDelay`) permettent de limiter la création d'hypothèses.

Deuxièmement, les contraintes temporelles (de type `Timeout` et `MaxDelay`) ainsi que l'opérateur `Without` permettent de supprimer des hypothèses existantes.

L'utilisation de ces contraintes introduit des coupures dans le processus de détection et peut mener à des faux négatifs mais c'est un choix explicite de celui qui écrit la signature. Elles permettent néammoins à l'algorithme de garder une complexité raisonnable en taille mémoire et en temps de calcul.

D'autres types de contraintes explicites sont envisageables mais ne sont pas, à l'heure actuelle, implantées dans ADeLe.

Par exemple, on pourrait définir une contrainte sur des événements qui implique que les transitions associées font progresser les hypothèses sans les dupliquer auparavant : c'est-à-dire qu'on ne conserverait pas d'hypothèses intermédiaires sur le franchissement de ces transitions. Ce type de contrainte est présent dans le langage STATL [EVK00] (*consuming transitions*). Si aucune transition ne duplique les événements, alors on obtient au maximum une alerte par démarrage de signature.

On pourrait également envisager un mécanisme qui empêche la création d'une hypothèse dans l'automate s'il en existe déjà une autre qui comporte la même valuation pour un n-uplet particulier de variables d'unification (déclarées dans la section `<CONTEXT>`). Cela impliquerait de maintenir une liste globale de ces valuations. Ce type de contrainte est similaire aux «classes d'équivalence» du langage Sutekh [PD00] et à l'opérateur `Synchronized` du langage LogWeaver [**?**].

Chapitre 4

Mise en œuvre et expérimentation

Dans ce chapitre, nous présentons les réalisations logicielles et les expérimentations effectuées dans le cadre de cette thèse.

La section 4.1 est consacrée à la mise en œuvre de quelques capteurs (système et réseau) générant des événements et d'une sonde applicative générant des alertes.

La section 4.2 détaille la réalisation d'un compilateur du langage ADeLe vers le langage des automates d'ADeLaIDS.

La section 4.3 présente ADeLaIDS, prototype fonctionnel d'analyseur qui met en œuvre l'algorithme de détection présenté dans le chapitre 3 et dont le détail se trouve en annexe F.

4.1 Mise en œuvre de capteurs et de sondes

Dans cette section, nous décrivons respectivement la réalisation d'un capteur système sous Solaris, d'un capteur système sous Linux, d'un capteur réseau et d'une sonde applicative pour serveur Apache.

4.1.1 Capteur système : audit BSM sous Solaris

Dans le cadre du projet Mirador[1], nous avons réalisé un capteur dédié à l'audit système Solaris qui a été utilisé comme source d'information pour l'IDS G^NG(réalisé à Supélec).

Ce capteur, écrit en langage C, permet de fournir des événements pré-filtrés qui peuvent ensuite être corrélés pour détecter une attaque. La source de données

[1]projet d'Etude Amont financé par la DGA (1999-2002) impliquant Alcatel, l'ONERA, Supélec et l'ENST Bretagne.

analysée par le capteur est l'audit BSM du système d'exploitation Solaris. Le capteur renvoie des événements encapsulés dans des messages IDMEF [CD03].

Le capteur scrute le fichier de log contenant l'audit binaire. Chaque nouvel enregistrement est traduit au format texte grâce à la commande système *praudit*.

Chaque enregistrement d'audit texte contient le type d'événement système audité (`AUE_EXECVE`, `AUE_CREAT`, ...). Le capteur peut être configuré avec la liste des types qui l'intéressent pour effectuer un pré-filtrage.

Si l'enregistrement passe ce premier filtre, il y a extraction des valeurs contenues dans les différents champs. On construit alors un message au format IDMEF qui contient ces valeurs. Certaines valeurs sont mises dans des sous-champs des balises `Source` ou `Target` tandis que les autres sont mises dans des champs `AdditionalData`.

Le capteur prend également en entrée un fichier de règles où l'on définit les contraintes contextuelles que doit vérifier le message IDMEF obtenu à l'étape précédente. Exemple de règle :

```
MIR-160 ::= (Alert.Classification.name = "AUE_EXECVE")
         && (Alert.Source.Process.name = "/etc/security/audit")
```

Si le message IDMEF vérifie l'une des règles[2], alors le champ `Alert.Classification.name` prend pour valeur le nom de la règle (`MIR-160` dans notre exemple). Le message IDMEF correspondant est alors émis par le capteur. En annexe, la figure H.1 (p.162) montre un exemple d'événement fourni par le capteur.

Pour ce capteur, nous avons utilisé le format IDMEF qui n'est pas destiné à représenter des événements. La plupart des attributs de l'événement sont dans des champs `AdditionalData`. En outre, les autres valeurs qui ont pu être mises dans des champs habituels du message IDMEF ne respectent pas forcément la sémantique associée à ces champs. Toutefois, l'expérience menée avec G^NG a montré que la détection par corrélation de ces événements fonctionne dans la pratique.

4.1.2 Capteur système : Libsafe sous Linux

Afin de montrer l'intérêt du format EVMEF (cf annexe E), nous avons réalisé un capteur système sous Linux qui est une modification du code de la librairie Libsafe[3] [TS01], distribuée sous licence GPL.

[2] la première règle vérifiée est utilisée.

[3] http://www.research.avayalabs.com/project/libsafe/

Libsafe est une librairie dynamique qui remplace certaines fonctions de la glibc pouvant provoquer des débordements de tampons[4] par recopie (*strcpy, strcat, ...*) ou des erreurs de format[5] (*scanf, sprintf, ...*).

Le remplacement à l'exécution de ces fonctions est transparent pour les programmes surveillés. Elles peuvent être pré-chargées pour un seul programme (variable d'environnement `LD_PRELOAD`) ou globalement pour tous les programmes (référence dans le fichier `/etc/ld.so.preload`).

Lorsqu'un appel à l'une des fonctions tente d'écrire dans un emplacement illicite en mémoire, le processus est automatiquement interrompu et une trace est écrite via *syslog*. D'autres formes d'avertissement sont possibles comme l'envoi d'un mail ou la génération d'une alerte[6]IDMEF.

Dans notre version modifiée (qui représente environ 300 lignes de code supplémentaires en langage C), nous avons ajouté la possibilité d'émettre un événement système de type "libsafe" au format EVMEF. Notre code est fonctionnellement proche du code existant qui génère une alerte IDMEF. Toutefois, nous avons fait le choix de construire un événement plutôt qu'une alerte pour plusieurs raisons. Premièrement, l'alerte IDMEF qui peut être construite utilise les balises génériques `AdditionalData` car l'alerte ne dispose pas de champs appropriés. Deuxièmement, les valeurs générées pour identifier la sonde qui émet l'alerte changent à chaque alerte (car c'est le processus interrompu qui la construit), ce qui rend la corrélation plus difficile. Et finalement, il existe des cas où Libsafe peut détecter un problème sans qu'il y ait de tentative d'exploitation malicieuse (ex : erreurs de design dans l'utilisation des chaînes de caractères pour les logiciels traduits en plusieurs langues) ce qui ne justifie pas forcément l'envoi d'une alerte.

Lorsqu'une fonction modifiée détecte un problème, nous construisons un nœud XML de type `<libsafe>` qui contient les données brutes de l'événement. Cette portion de document XML est envoyé via UDP sur un port dédié de la machine locale. Un démon en écoute sur ce port peut alors encapsuler ces données dans le corps d'un message EVMEF. L'utilisation d'un processus unique pour la construction des messages permet d'avoir une identification unique du capteur (champ `System/Analyzer@analyzerid`). Ce message, dont le champ `System/Classification/name` porte la valeur "Libsafe_event", peut alors être émis par le capteur.

En annexe, la figure H.2 (p. 163) présente un exemple d'événement Libsafe.

4.1.3 Capteur réseau : sniffer TCP/UDP/ICMP

Nous avons réalisé un capteur réseau, écrit en langage C, qui génère des événements réseau au format EVMEF (voir annexe E). Le format utilisé pour repré-

[4] *buffer overflow*

[5] *format string bugs*

[6] une sonde pour l'IDS Prelude est intégrée au code de Libsafe, http://www.prelude-ids.org

senter les données brutes est donné sous la forme d'une grammaire en annexe C.

Ce capteur utilise la librairie de capture de paquets Libpcap[7]. Il prend en entrée des paquets réseau dont les trames sont au format Ethernet II. Les paquets peuvent être capturés directement sur une interface réseau ou lus à partir d'un fichier au format Tcpdump.

La mise en œuvre de ce capteur a consisté à écrire une fonction de *callback* qui décode en XML les paquets qui ont passé le filtre associé à la Libpcap. Cette fonction contient un arbre de décision qui permet le décodage des couches encapsulées (construit à partir des RFC afférentes à Ethernet, ARP, TCP, UDP, ICMP et RPC). Ces données brutes sont contenues dans une balise XML de type `<packet>`, laquelle est ensuite incluse dans une en-tête EVMEF de type `<Network>`.

Le capteur se configure en entrée avec un nom d'événement (ex : `rpcinfo`) et un filtre au format BPF[8] qui est utilisé pour initialiser la Libpcap. Ainsi, chaque paquet qui passe le filtre donne lieu à la création d'un événement EVMEF de type réseau.

Des tests de temps de décodage *off-line* ont été réalisés sur une machine disposant d'un processeur Athlon à 600 Mhz. Tous les paquets capturés dans un fichier sont décodés en XML (sans l'entête EVMEF). La vitesse de décodage est d'environ 2.6 Mo de paquets binaires par seconde (avec un filtre nul, c'est-à-dire tous les paquets décodés). D'autre part, le rapport de taille entre les données brutes décodées en XML et les paquets binaires est d'environ 4 (testé sur plusieurs centaines de méga-octets).

En annexe, la figure H.3 (p.164) montre un exemple d'un événement réseau au format EVMEF, nommé "`rpcinfo`".

La fonction de décodage contenue dans le capteur permet de visualiser un paquet réseau (de type Ethernet) sous une forme compréhensible par un humain. Elle peut être réutilisée dans d'autres logiciels. Par exemple, on peut envisager une modification de l'IDS Snort[9] afin que les alertes qu'il génère contiennent les paquets décodés en XML. On pourrait utiliser cette fonction pour ajouter à l'utilitaire *tcpdump* un nouveau format pour «dumper» les paquets.

[7] http://www.tcpdump.org/
[8] Berkeley Packet Filter
[9] http://www.snort.org

4.1.4 Sonde applicative : module pour serveur web Apache

Dans le cadre du projet Mirador, nous avons réalisé une sonde applicative pour serveur web Apache[10] (écrite en langage C). Elle permet, par recherche de motifs connus dans les requêtes, de détecter des attaques contre des scripts vulnérables du serveur web. Elle est basée sur la version 1.0 de "Mod_id"[11] programmée par Burak Dayioglu.

La sonde se présente sous la forme d'un module Apache. La sonde est configurable à partir d'une liste de motifs qui sont recherchés pour chaque URL accédée sur le serveur (au moment de la phase de logging d'Apache). Ces motifs sont contenus dans un fichier de configuration sous forme d'expressions régulières. Ils sont caractéristiques d'attaques connues (cgi-bin et autres).

Exemples de motifs :

```
^/cgi-bin/phf
^/cgi-bin/htsearch\?exclude
```

La version originale de "Mod_id" comparait successivement l'URL accédée avec chacun des motifs (représenté par une chaîne de caractères constante) à l'aide de la fonction standard *strncmp* et générait un message via syslog en cas de détection.

Dans notre version, nous avons utilisé la librairie "regex" standard pour effectuer une recherche efficace de tous les motifs à la fois. A partir de la liste des motifs, nous avons construit une expression régulière étendue qui réalise un OU entre tous ces motifs. Ce motif de recherche global est alors pré-compilé une seule fois à l'initialisation du module (c'est à dire au démarrage du serveur web). Le regroupement en une seule expression permet de tirer parti d'éventuels préfixes communs.

Nous avons également programmé la création d'une alerte au format IDMEF (version 0.3) à chaque détection. Chacune de ces alertes comporte le même nom générique "Suspicious URL" (dans le champ `Alert/Classification[0]/name`).

En annexe, la figure H.4 (p. 165) montre un exemple d'alerte générée par la sonde.

Plusieurs extensions et améliorations peuvent être envisagées. Afin de limiter les possibilités d'évasion par camouflage d'URL (ex : codage hexadécimal), il faut mettre en place un processus de normalisation des requêtes reçues par le serveur avant d'y rechercher les motifs. On pourrait également préciser le diagnostic de l'alerte en déterminant la réussite ou l'échec de la requête HTTP (grâce aux codes de retour).

[10] http://www.apache.org

[11] http://www.dayioglu.net/projects/mod_id_1.0.tar.gz

4.2 Compilation du langage ADeLe vers le langage des automates d'ADeLaIDS

Nous avons réalisé un compilateur pour traduire une spécification ADeLe en un automate de reconnaissance correspondant à la partie détection. Le résultat de la compilation permet d'obtenir un fichier texte qui contient la représentation équivalente en termes d'états, de transitions, de définitions d'événements et de contraintes temporelles et contextuelles. C'est cette représentation intermédiaire qui sera utilisée à l'initialisation de l'analyseur ADeLaIDS.

Nous avons choisi d'utiliser le compilateur de compilateurs ANTLR[12] et ce pour deux raisons. Tout d'abord, la notation EBNF[13] permet d'exprimer de façon naturelle les répétitions à l'aide des opérateurs '*' et '+' (contrairement au couple Flex/Bison où l'on doit utiliser la récursivité pour les exprimer). Ensuite, ANTLR peut produire un compilateur écrit en Java[14]. Cela nous permet de l'interfacer (voire de l'intégrer) plus facilement avec l'analyseur ADeLaIDS qui est lui-même écrit en Java.

La grammaire EBNF complète du langage ADeLe est disponible en annexe C. Nous ne détaillons ici que la transformation de la partie détection du langage, c'est-à-dire tout ce qui est englobé par la balise `<DETECTION>` dans une description. Les autres parties de la description ne sont en effet pas traduites, mais seulement validées par rapport à la grammaire.

Nous présentons maintenant les différentes phases de la transformation d'une signature exprimée en ADeLe en un automate à états finis permettant d'en assurer la détection.

Extraction des définitions de types et d'instances d'événements

Dans la partie `<EVENTS>`, les actions incluses au sein de la grammaire nous permettent de construire différentes tables de symboles (noms des événements, type des événements et source de donnée associée). On réalise également des vérifications sémantiques qui ne peuvent pas être exprimées par la grammaire (ex : définition unique des noms de type). On conserve la définition des filtres associés à chaque type d'événement (afin de permettre le typage des événements bruts).

[12] *ANother Tool for Language Recognition*, http://www.antlr.org
[13] Extended Backus-Naur Form
[14] Le compilateur JavaCC le permet également (http://javacc.dev.java.net).

```
<ENCHAIN>
 (E0 ; Non_ordered{E1 , E2} ; E3 ; One_among{E4 , E5} ; E6)
</ENCHAIN>
```

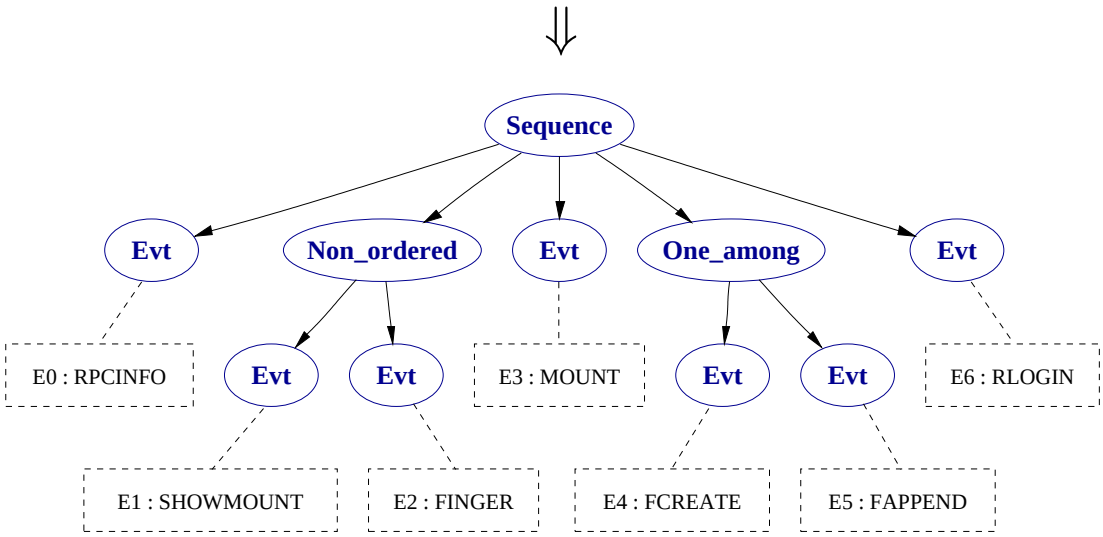

FIG. 4.1 – Arbre de syntaxe abstrait correspondant à la règle *scenario*

Construction d'un arbre correspondant à la signature temporelle

Dans la partie `<ENCHAIN>`, l'analyse grammaticale construit un arbre syntaxique abstrait correspondant à la règle *scénario*, comme le montre l'exemple de la figure 4.1. C'est un arbre n-aire dont chaque nœud intermédiaire représente un type d'opérateur : `Sequence`, `One_among`, `Non_ordered` ou `Without`. Les feuilles de l'arbre représentent les événements élémentaires (`Evt`). C'est le parcours de cet arbre qui va nous permettre de créer les états et les transitions de notre automate de reconnaissance.

S'il y a des contraintes temporelles supplémentaires (`Timeout`, `MinDelay` ou `Maxdelay`), elles sont également extraites et conservées.

Extraction des contraintes contextuelles

Dans la partie `<CONTEXT>`, toutes les contraintes contextuelles sont extraites et transformées en gardes qui seront portées par les transitions correspondantes (cf paragraphe 3.3, p. 70+). Pour chaque champ d'un événement qui intervient dans une contrainte contextuelle, on conserve une association entre le type de l'événement et le champ référencé.

Recensement des champs à extraire pour construire les alertes

Dans la partie `<REPORT>`, certaines lignes référencent une valeur extraite d'un champ d'un événement particulier. Comme pour les contraintes contextuelles, on conserve une association entre le type de l'événement et le champ référencé.

Création et assemblage des états et des transitions

On parcourt en profondeur l'arbre de syntaxe abstrait afin d'en déduire la liste des états et des transitions de l'automate de reconnaissance. Les sous-automates élémentaires correspondent aux feuilles de l'arbre. Ils sont assemblés, lors de la remontée dans l'arbre, par la traduction des opérateurs en états et transitions supplémentaires (cf paragraphe 3.3, p.70+).

Chaque feuille de l'arbre (figure 4.2) est de type `Evt`. On représente la reconnaissance d'un seul événement par un sous-automate comportant deux états reliés par une transition. La transition est en attente d'un événement (comportant un `TYPE` associé et éventuellement un `NOM` d'instance). Elle référence ses états d'origine et de destination. L'état en amont de la transition est destiné à contenir des hypothèses : il est de type `NORMAL`.

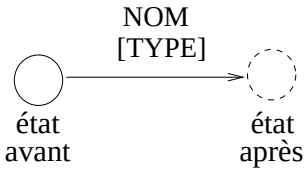

FIG. 4.2 – Reconnaissance d'un événement

Dans l'arbre abstrait, un nœud de type `Sequence` permet d'assembler des sous-automates qui doivent être enchaînés (correspondant aux sous-arbres de type `Evt`, `One_among`, `Non_ordered` ou `Without`). On voit dans la figure 4.3 que cela consiste simplement à fusionner l'état de fin d'un sous-automate avec l'état de début du suivant. Il n'y a pas de création d'état ou de transition supplémentaire.

Dans l'arbre abstrait, un nœud de type `One_among` représente une alternative entre plusieurs sous-automates (figure 4.4). On crée d'abord un nouvel état (de type `MULTI`) qui représente l'état avant reconnaissance de l'automate associé à l'opérateur `One_among`. Ensuite, il y a création de plusieurs ε-transitions qui relient ce premier état aux états de début[15] de chacun des sous-automates. Finalement, les états de fin de chacun des sous-automates sont fusionnés en un

[15] les états de début étant éventuellement de types différents, ils ne peuvent pas être fusionnés.

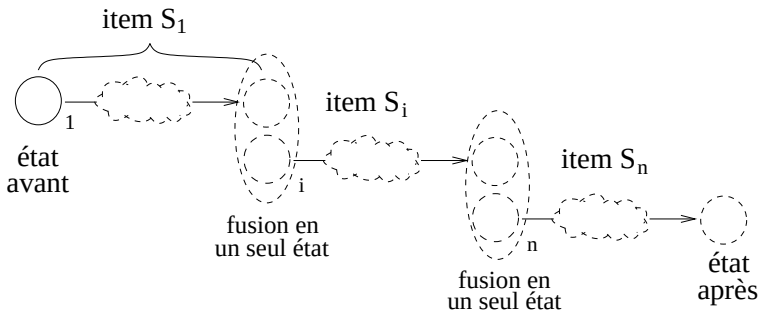

FIG. 4.3 – Opérateur de Séquence

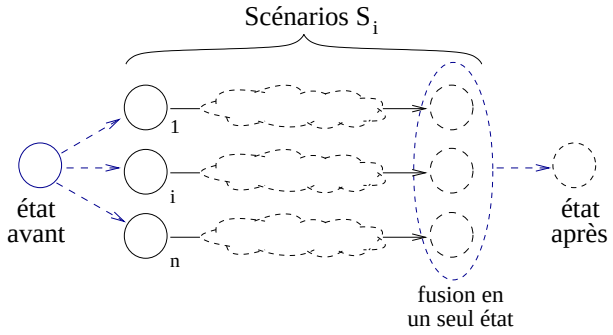

FIG. 4.4 – Opérateur One_among

seul (de type `MONO`). Une dernière ε-transition relie cet état à celui qui représente l'état après reconnaissance de l'automate.

Dans l'arbre abstrait, un nœud de type `Non_ordered` représente une conjonction de plusieurs sous-automates (figure 4.5). On procède de la même façon que pour un nœud de type `One_among`. Toutefois, le premier état généré est de type `DISTRIB` et l'état fusionné est de type `MERGE`.

Dans l'arbre abstrait, un nœud de type `Without` comporte deux sous-arbres qui représentent les sous-automates associés respectivement à la branche positive et à la branche négative d'un opérateur `Without`. On crée un nouvel état (de type `WITHOUT`) qui représente l'état avant reconnaissance de l'automate associé à l'opérateur `Without`. On crée ensuite une ε-transition de cet état vers l'état de début du sous-automate «positif». On fusionne également les états de fin des deux sous-automates en un seul état (de type `SYNC`). Nous rappelons que le sous-automate «négatif» sera activé par une transition du sous-automate «positif». Une dernière ε-transition est créée pour relier cet état à l'état après reconnaissance de l'automate.

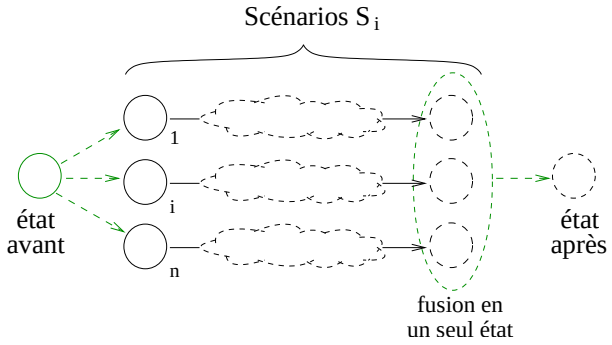

FIG. 4.5 – Opérateur Non_ordered

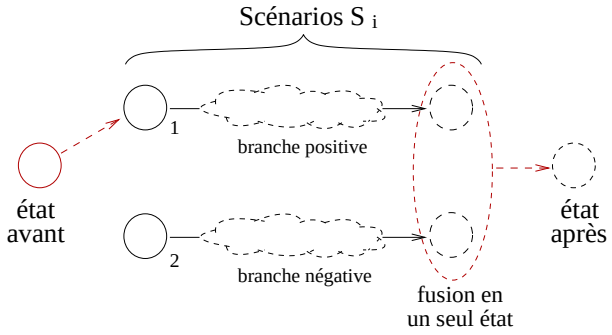

FIG. 4.6 – Opérateur Without

Génération de l'automate et du graphe associé

La dernière étape de la compilation consiste à générer la représentation textuelle de l'automate complet (c'est-à-dire comportant toutes les informations extraites lors de l'analyse grammaticale) ainsi que le graphe associé au format texte de Graphviz[16] (utilisé par l'interface graphique d'ADeLaIDS pour visualiser les automates).

En annexe, la figure G.1 (p. 158) présente le résultat de la compilation de la description d'attaque `"NFS_Mount"`. C'est la représentation qui est lue par l'analyseur ADeLaIDS pour construire ses automates de reconnaissance.

Elle comporte le nom de l'attaque à reconnaître (`NAME`), le numéro d'état initial de l'automate (`INIT`) ainsi qu'un éventuel timeout global (`TIMEOUT`). Les cinq sections qui suivent représentent respectivement la liste des états (`STATES`), la liste des transitions (`TRANSITIONS`), la liste des gardes (`CONSTRAINTS`), la liste des filtres associés à chaque type d'événement (`FILTERS`) et la liste des champs utiles d'un événement filtré (`EXTRACT`).

[16] http://www.graphviz.org

4.3 L'analyseur ADeLaIDS

4.3.1 Principe de fonctionnement

L'analyseur ADeLaIDS recherche des signatures dans un flux d'événements en s'appuyant sur des automates de reconnaissance (générés par compilation d'une spécification ADeLe).

L'architecture visée est représentée dans la figure 4.7.

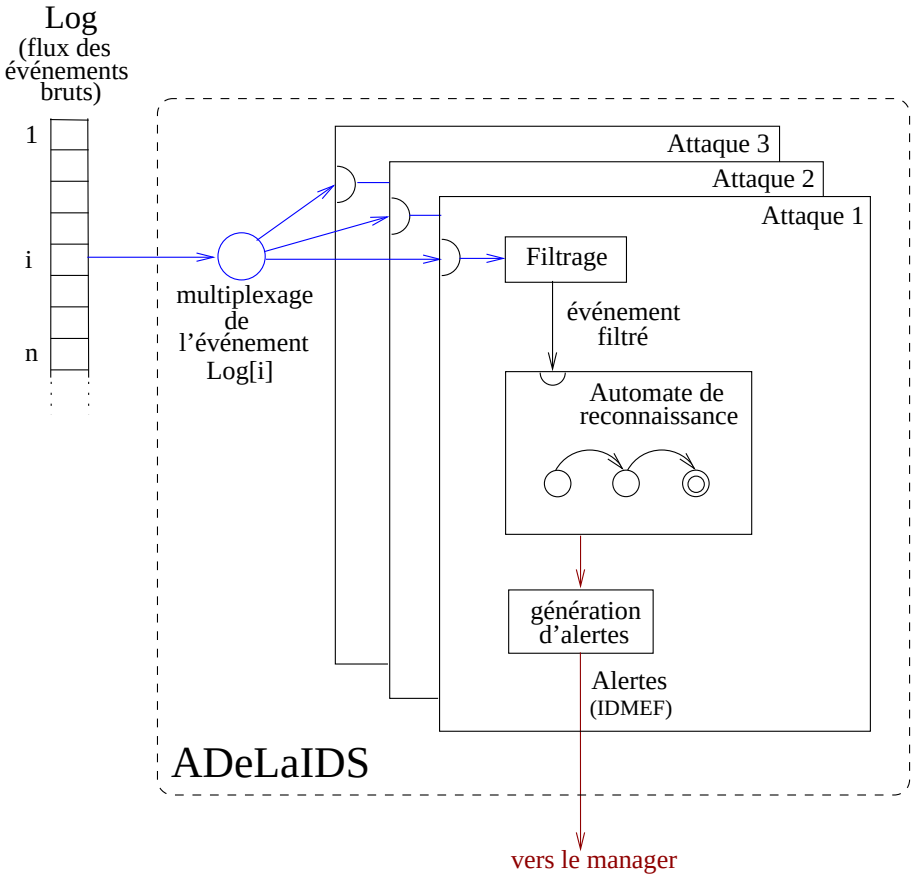

FIG. 4.7 – Principe de fonctionnement d'ADeLaIDS à l'exécution

Dans la version courante de la maquette, le coeur du moteur d'analyse, qui prend en entrée des événements filtrés, est fonctionnel[17]. La fonction de filtrage locale à chaque signature est externalisée en Perl (adapté à la manipulation de texte). D'autre part, les tests de la maquette n'ont été effectués qu'avec un seul automate à la fois.

[17] l'opérateur Without n'est pas implanté (pas de démarrage des automates inhibiteurs).

4.3.2 Mise en œuvre

L'algorithme de détection présenté dans le chapitre 3 est adapté à une implantation dans un langage orienté objets. Nous avons choisi le langage Java pour sa mise en œuvre. La plupart des types abstraits de données de l'algorithme sont directement traduits en une classe Java correspondante.

Dans la figure 4.8 (p. 105), nous avons représenté le diagramme de classes UML qui précise les principales relations existant entre les différentes classes Java. On note que la classe Adelaids relie une instance de Log à une ou plusieurs instances d'Automate. Les classes dérivées de la classe Etat redéfinissent la méthode `ajoutHypothese`, ce qui permet d'effectuer des actions différentes lorsqu'une instance d'Hypothèse arrive dans une instance d'Etat.

ADeLaIDS a deux modes de fonctionnement :
- un mode console qui analyse en *off-line* les événements d'un fichier de Log par rapport à des signatures ;
- un mode graphique interactif de test d'automate qui prend en entrée un fichier contenant des événements pré-filtrés, une signature et le graphe associé.

4.3.3 Interface graphique interactive de test des automates

Dans la figure 4.9, nous avons représenté l'interface graphique d'ADeLaIDS utilisée lorsqu'il est lancé en mode interactif. La signature utilisée dans cet exemple est celle de l'attaque "NFS_Mount" (dont la description est en annexe G.1, p. 153).

L'interface comporte les éléments suivants :
- la représentation du graphe associé à l'automate de reconnaissance (utilisation de Grappa[18], l'API java de Graphviz) ;
- le menu déroulant permettant de choisir l'événement envoyé à l'automate (provenant du fichier fourni en entrée) ;
- la fenêtre texte qui affiche les traces de l'exécution (comme dans le mode console).

Ce mode de fonctionnement interactif permet de tester et de visualiser les automates. L'horodatage de l'événement envoyé à l'automate lorsque l'on clique sur un bouton est l'heure courante de la machine (et non pas celui référencé dans le fichier des événements d'entrée). Ce fonctionnement en horloge réelle permet notamment d'expérimenter les effets des contraintes temporelles (`Timeout`, `MinDelay` et `MaxDelay`).

[18] API java permettant de visualiser des graphes au format Graphviz, http://www.research.att.com/ john/Grappa/

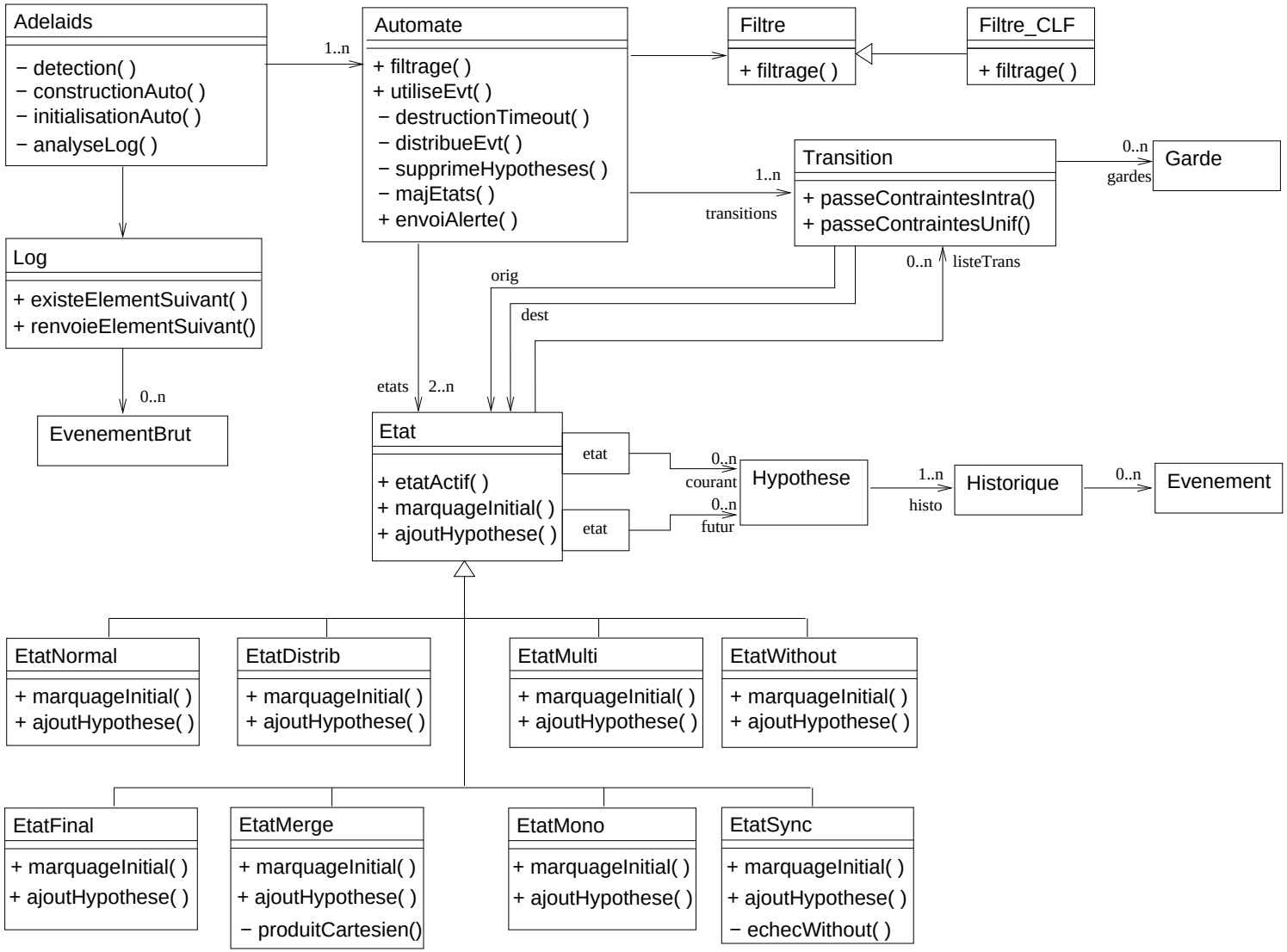

FIG. 4.8 – Diagramme de classes UML

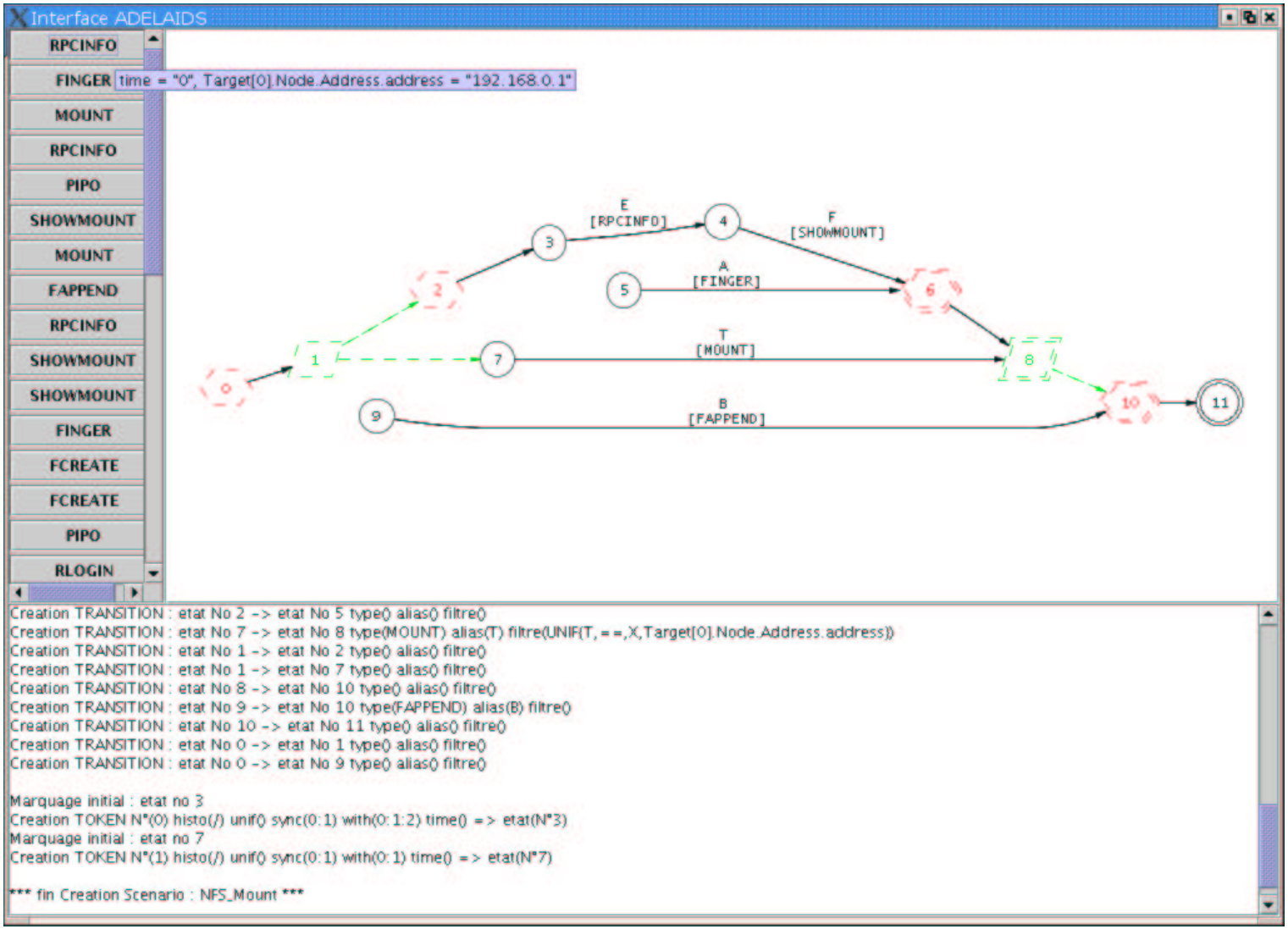

FIG. 4.9 – Interface graphique d'ADeLaIDS en mode interactif

4.3.4 Expérimentation

4.3.4.1 Test d'un cas réel : analyse d'un log Apache

Nous avons expérimenté l'analyse d'un log réel de serveur web Apache. Nous avons écrit une signature (figure 4.10) permettant de détecter un scan CGI effectué par un outil automatique de tests de vulnérabilité. Elle consiste à rechercher trois motifs caractéristiques dont on trouve généralement les traces dans les logs d'accès du serveur web cible. Ce genre d'outil génère souvent des centaines de requêtes dans un temps très court. Nous recherchons donc ces trois motifs dans n'importe quel ordre (opérateur `Non_ordered`) et sur une durée maximum de 60 secondes.

```
<DETECTION>
  <DETECT>
    <EVENTS>
      APACHE : CLF {Appli/Classification[0]/name == "CLF";
      } A0, A1, A2;
    </EVENTS>
    <ENCHAIN>
     Non_ordered{A0, A1, A2}
     Timeout(60)
    </ENCHAIN>
    <CONTEXT>
      A0/Clf/url MATCHES "^.*/etc/passwd.*$"
      A1/Clf/url MATCHES "^.*perl\.exe.*$"
      A2/Clf/url MATCHES "^.*htsearch.*$"
      X := A0/Clf/ip
      A1/Clf/ip == X
      A2/Clf/ip == X
    </CONTEXT>
  </DETECT>
  ...
</DETECTION>
```

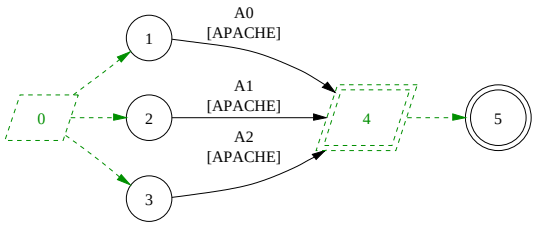

FIG. 4.10 – Signature détectant un scan CGI

Les logs utilisés proviennent d'un site web interne de Supélec. Ils représentent un "access_log" de 49803 lignes correspondant à une période de 11 mois.

N.B. : l'analyse a été réalisée sur un PC (Pentium3 450 Mhz, 192 Mo RAM) sous Linux (noyau 2.4.20). La machine virtuelle Java utilisée est la version 1.4.1 (mémoire allouée de 128 Mo maximum).

Le temps de traitement du log par ADeLaIDS est de 14 secondes, ce qui est acceptable pour l'analyse de près de 50000 lignes. Le résultat de l'analyse (figure 4.11) montre qu'il y a deux pics dans le nombre d'hypothèses présentes au sein de l'automate.

Au total, 52 alertes ont été générées durant ces deux pics. Elles correspondent effectivement dans les logs à des scans CGI effectués contre ce serveur web en décembre 2002 et janvier 2003.

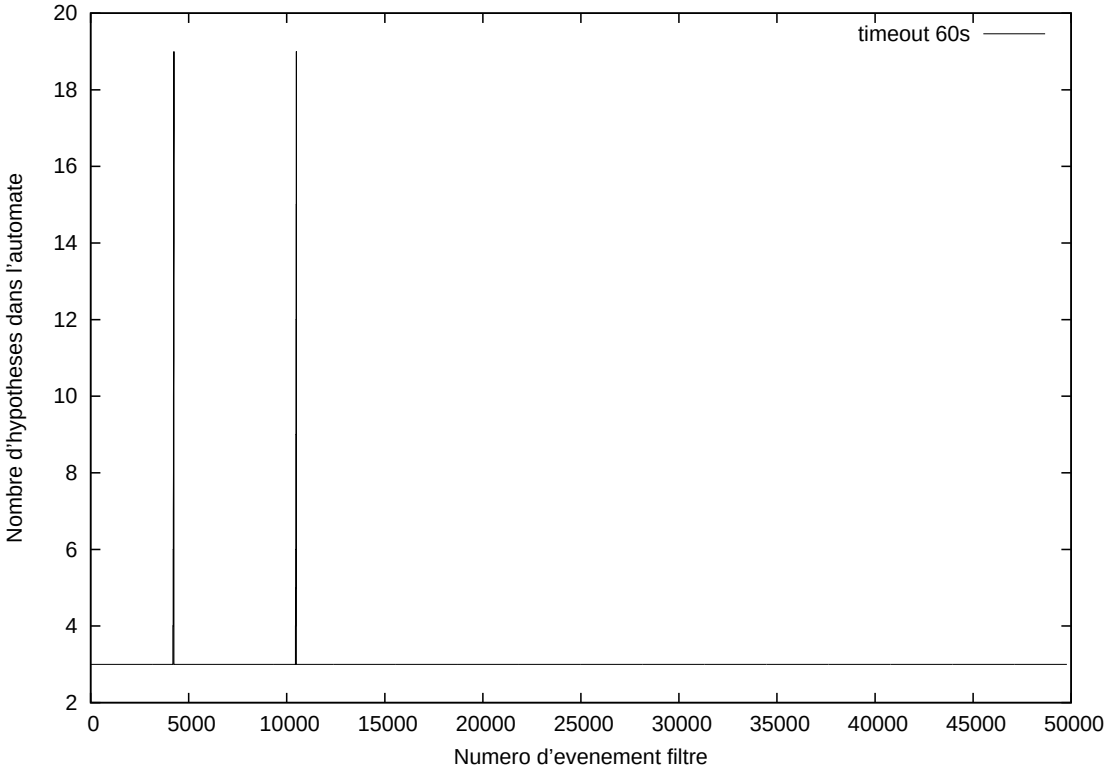

FIG. 4.11 – Analyse d'un log Apache à la recherche d'un scan CGI

4.3.4.2 Test d'un cas limite par simulation

Afin de mesurer le phénomène d'explosion combinatoire, nous avons écrit une signature volontairement complexe (qui ne correspond pas à une signature connue). Le but de cette expérience est de placer l'algorithme de détection dans un cas défavorable.

La signature utilisée est présentée dans la figure 4.12. Le log utilisé est constitué de 144 événements dont les horodatages sont répartis sur un intervalle de 14 se-

condes. Afin de déclencher le maximum de transitions dans l'automate, les valeurs des champs des événements sont toutes compatibles avec les contraintes intra-événement (`File_Modified/name` et `File_Created/name`) et inter-événements (`Target[0]/Node/Address/address`).

Une série de quatre tests a été réalisée avec des durées de Timeout différentes. Les résultats obtenus sont représentés sur la figure 4.13 qui donne le nombre d'hypothèses présentes au sein de l'automate après la prise en compte de chaque événement. L'échelle utilisée en ordonnée est logarithmique.

N.B. : les mesures ont été réalisées sur un PC (Athlon 600 Mhz, 256 Mo RAM) sous Linux (noyau 2.4.21). La machine virtuelle Java utilisée est la version 1.4.1 (mémoire allouée de 128 Mo maximum).

Aucun Timeout
La progression de la première courbe est exponentielle. On voit qu'elle s'arrête après l'événement numéro 71 (189553 hypothèses présentes). L'analyse s'est interrompue après 2 mn 37 s de calculs : l'analyseur a saturé la mémoire. Avant l'interruption, il a pu émettre un nombre total de 92872 alertes.

Timeout de 10 s
En utilisant un Timeout de 10 secondes, on obtient exactement le même résultat : la deuxième courbe est confondue avec la première. Le temps de calcul et le nombre d'alertes générées est identique au test précédent. La saturation s'est produit avant que le Timeout de 10 secondes produise ses effets.

Timeout de 4s
Lorsque le Timeout est de 4 secondes, l'analyse traite l'intégralité du log en 7 secondes. La courbe montre des périodes de progression exponentielles qui sont entrecoupées par des chutes du nombre d'hypothèses dues au Timeout. L'analyse a généré 11604 alertes.

Timeout de 1s
Lorsque le Timeout est de 1 secondes, l'analyse traite l'intégralité du log en 2 secondes. La courbe se comporte de façon similaire à la précédente mais avec une périodicité plus courte et à une échelle moindre en nombre d'hypothèses. L'analyse a généré 316 alertes.

```
<DETECTION>
  <DETECT>
    <EVENTS>
      RPCINFO   : PACKET { Network/Classification[0]/name == "rpcinfo" ;   } E0 ;
      SHOWMOUNT : PACKET { Network/Classification[0]/name == "showmount" ; } E1 ;
      FINGER    : PACKET { Network/Classification[0]/name == "finger" ;    } E2 ;
      MOUNT     : PACKET { Network/Classification[0]/name == "mount" ;     } E3 ;
      FAPPEND   : SNARE  { System/Classification[0]/name  == "fappend" ;   } E4 ;
      FCREATE   : SNARE  { System/Classification[0]/name  == "fcreate" ;   } E5 ;
      RLOGIN    : PACKET { Network/Classification[0]/name == "rlogin" ;    } E6 ;
      EXTRA     : PACKET { Network/Classification[0]/name == "extra";      } E7,E8;
    </EVENTS>
    <ENCHAIN>
      ( E0 ; Non_ordered{(E1 ; E2), Non_ordered{E3,E4}, [(E5;E6)] Without{E7}} ; One_among{E8,(EXTRA^5)} )
    </ENCHAIN>
    <CONTEXT>
       # contraintes intra-evenement
      E4/File_Modified/name == ".rhosts"
      E5/File_Created/name == ".rhosts"
       # contraintes inter-evenement
      X := E0/Target[0]/Node/Address/address
      E1/Target[0]/Node/Address/address == X
      E2/Target[0]/Node/Address/address == X
      E3/Target[0]/Node/Address/address == X
      E4/Target[0]/Node/Address/address == X
      E5/Target[0]/Node/Address/address == X
      E6/Target[0]/Node/Address/address == X
    </CONTEXT>
  </DETECT>
  ...
</DETECTION>
```

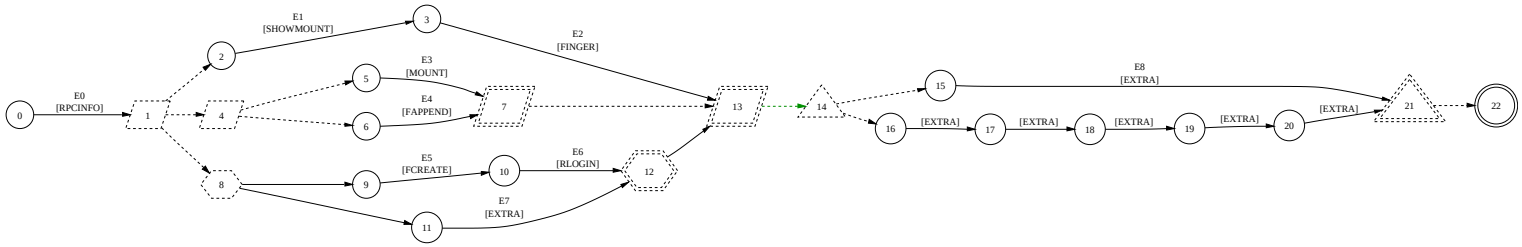

FIG. 4.12 – Signature complexe

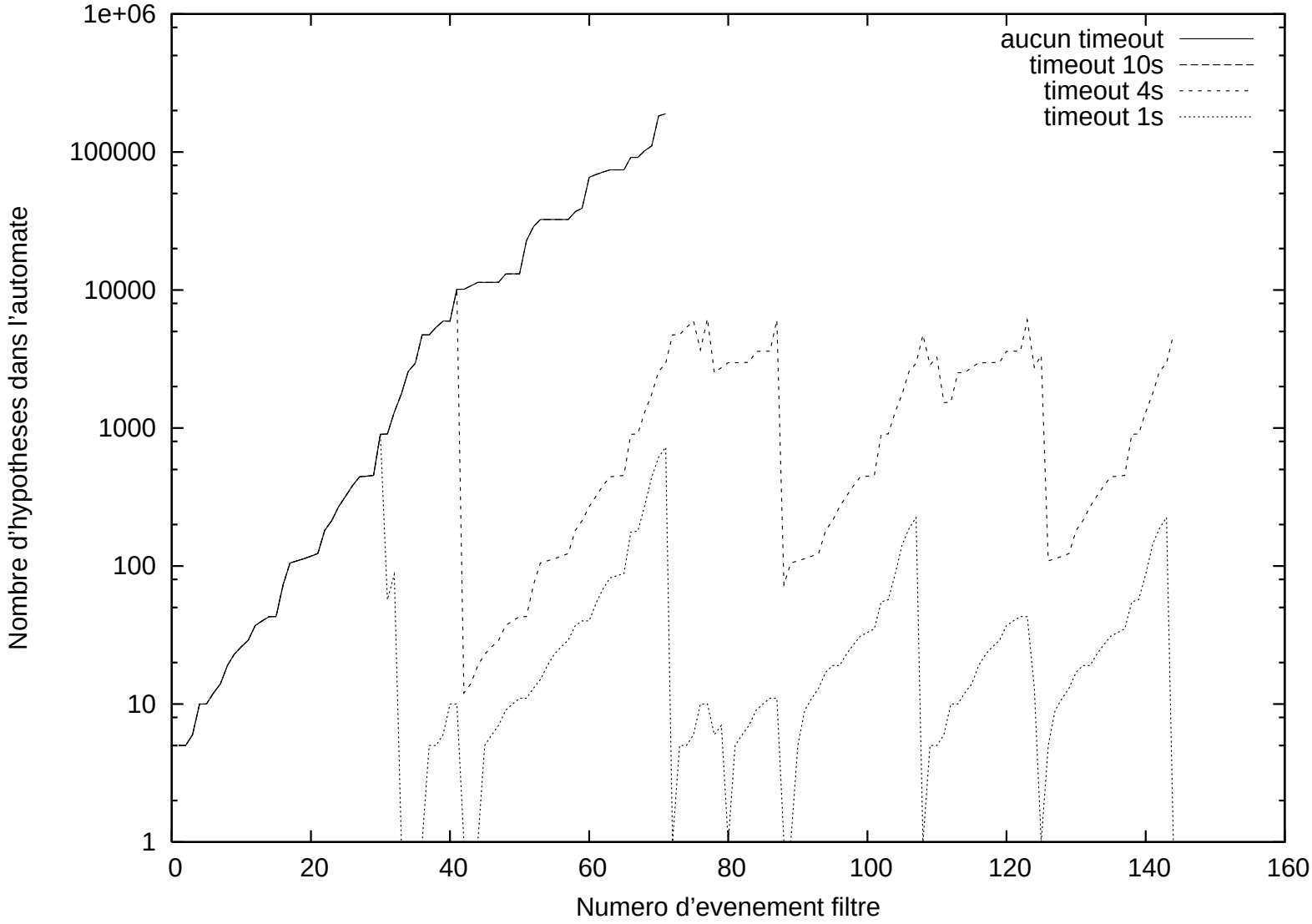

FIG. 4.13 – Mesures pour différentes valeurs du Timeout (échelle logarithmique)

Nous concluons de cette expérience que la coupure temporelle que constitue le Timeout est nécessaire pour limiter la saturation de la mémoire lors de l'analyse. La durée utilisée pour le Timeout doit donc être un compromis entre le risque de saturation et le risque de provoquer des faux négatifs.

Retour d'expérience

L'expérience de mise en œuvre de capteurs nous a permis de relever des problèmes récurrents sur la qualité des audits systèmes :

- les formats de trace sont souvent mal définis (ex : absence de caractères d'échappement pour les délimiteurs de champs dans l'audit BSM[19] pour Solaris et dans l'audit Snare[20] pour Linux) ;
- les API pour exploiter directement l'audit binaire natif font souvent défaut (ex : transformation préalable obligatoire au format texte pour l'audit BSM et pour l'audit Windows) ;
- certaines informations ne peuvent pas être auditées (ex : certains attributs de fichiers et les variables d'environnement dans l'audit Snare).

[19]SunSHIELD Basic Security Module Guide , http://docs.sun.com/db/doc/802-5757
[20]http://www.intersectalliance.com/projects/Snare/

Conclusion et perspectives

Dans ce mémoire, nous avons présenté, dans le chapitre 1, un aperçu du domaine de la détection d'intrusions. La présentation d'une classification des IDS et des différents types de langages utilisés pour décrire une attaque permet de situer nos travaux dans le contexte de l'approche par scénarios.

Dans le chapitre 2, nous avons détaillé la syntaxe et la sémantique informelle associées au langage ADeLe. Ce langage donne les moyens de décrire une attaque sous toutes ses facettes : exploit, détection et réaction. La partie détection permet de corréler des événements ou des alertes en définissant des contraintes temporelles et logiques.

Dans le chapitre 3, nous avons défini la sémantique opérationnelle associée à la partie détection du langage. Nous présentons d'abord le formalisme (basé sur des automates à états finis) utilisé pour modéliser la détection. Nous détaillons ensuite, pour chaque élément du langage, la sémantique opérationnelle qui lui est associée dans notre modèle. Nous donnons également l'algorithme abstrait qui explicite le processus de détection (fondé sur l'analyse d'un flux d'événements par l'automate de reconnaissance associé à la signature). Nous abordons le problème de l'explosion combinatoire lié à notre modèle qui conserve en cours d'exécution, par défaut, l'ensemble des solutions partielles en mémoire.

Dans le chapitre 4, nous avons présenté différentes réalisations effectuées au cours de cette thèse. Cela inclut le développement de plusieurs capteurs système et réseau ainsi qu'une sonde applicative. Nous présentons également un compilateur du langage ADeLe vers celui de nos automates à états finis, ainsi que le prototype fonctionnel d'analyseur ADeLaIDS utilisant ces automates (qui représentent les signatures). Nous présentons également deux expérimentations effectuées à l'aide de cet analyseur.

Le premier objectif de cette thèse (cf page 4) était de fournir un langage de haut niveau d'abstraction permettant de décrire complètement une attaque, que ce soit sous l'angle de vue de l'attaquant ou de celui du défenseur. Cet objectif est atteint et l'on dispose désormais d'une grammaire complète du langage ainsi que d'une sémantique informelle. Cela constitue la première contribution de ce travail de thèse.

Le deuxième objectif (qui constitue un sous-ensemble du premier) était de fournir un langage de détection par corrélation d'événements ou d'alertes. Cet objectif est atteint : on dispose désormais d'une sémantique opérationnelle et d'un analyseur (configurable dans ce langage) permettant de réaliser la détection. Cela constitue la deuxième contribution de ce travail de thèse. On notera que le langage permet de traiter de façon identique les événements et les alertes. Ainsi, si la corrélation d'événements ne s'avère pas possible dans certains contextes pour des raisons de performances, on peut quand même faire de la corrélation d'alertes.

En revanche, nous n'avons pas effectué suffisamment de tests pour évaluer quantitativement la diminution espérée des faux-positifs liée à l'utilisation de la corrélation dans le processus de détection.

Dans le prolongement de ce travail de thèse, nous pouvons envisager plusieurs pistes pour des travaux futurs.

Il nous semble intéressant de réaliser une nouvelle implantation plus efficace de l'analyseur ADeLaIDS, dont le prototype actuel est écrit en Java. Premièrement, il serait raisonnable de l'implanter dans un autre langage permettant notamment une gestion plus fine de la mémoire. Deuxièmement, une modification de la sémantique opérationnelle (au niveau de la génération des automates et du modèle de franchissement des transitions) devrait permettre d'éviter la création artificielle d'hypothèses lors de l'utilisation des opérateurs `Non_ordered` et `One_among`.

La mise en œuvre concrète d'un langage adapté à l'écriture d'exploits systèmes et réseaux (tel que EDL présenté dans la section 2.2.2) nous semble particulièrement utile pour le test des IDS. Premièrement, cela permettrait d'avoir un langage de référence commun, ce qui éviterait de devoir utiliser différents compilateurs/interpréteurs pour exécuter l'attaque. Deuxièmement, cela permettrait d'avoir une alternative open-source aux outils commerciaux d'aide aux tests de pénétration qui commencent à voir le jour (par exemple Core Impact[21] et CANVAS[22]).

Nous envisageons des extensions au langage ADeLe lui-même. Nous proposons de rajouter dans ADeLe des opérateurs permettant de limiter la génération d'hypothèses (cf section 3.6) mais qui risquent en contrepartie de provoquer des faux négatifs. Par exemple, il est souhaitable d'introduire un opérateur similaire aux «classes d'équivalence» présentes dans le langage Sutekh [PD00]. Concrètement, cela revient à interdire la génération de plus d'une hypothèse possèdant une certaine valuation pour un n-uplet donné de variables (qui correspondent à des champs d'événements). On peut également introduire l'équivalent des «*consuming transitions*» présentes dans le langage STATL [EVK00]. Sur ce type de transitions, l'hypothèse candidate est consommée ; c'est-à-dire qu'elle n'est pas

[21] http://www.coresecurity.com/products/coreimpact/
[22] http://www.immunitysec.com/CANVAS/

dupliquée dans l'état en aval de la transition avant de progresser vers l'état suivant (avec augmentation de son historique). Cela implique que l'on ne n'explore pas l'intégralité des détections possibles, ce qui permet de lutter contre l'explosion combinatoire mais au risque de provoquer des faux-négatifs. Enfin, nous proposons de rajouter aux descriptions une contrainte à caractère topologique (portant sur le positionnement relatif des capteurs/sondes ayant généré les événements). Cela permettrait, par exemple, d'éviter de chercher à corréler entre eux des événements provenant de plusieurs réseaux différents alors même que l'attaque exige que ces événements soient générés sur le même réseau local.

Finalement, une piste essentielle pour prolonger ces travaux est la réalisation de tests significatifs afin de répondre aux questions restées en suspens : l'évaluation de la diminution des faux-positifs et l'évaluation des performances de la corrélation dans les sondes (en termes de temps de détection et en taille mémoire nécessaire). Cela passe par la définition d'une méthodologie rigoureuse de tests. On peut, à cet effet, s'inspirer de l'étude sur les tests d'IDS réalisée dans [MHL+03]. Cela passe également par la constitution de jeux de données de référence (dans lesquelles on sait précisément où se trouvent les activités intrusives et les activités légitimes) car il est difficile d'obtenir des données réelles (logs et trafic réseau) qui soient publiquement partagées.

Nous avons pu expérimenter quelques faiblesses de la détection d'intrusions basée sur l'approche par scénarios : la difficulté à détecter toutes les variantes possibles d'une attaque ainsi que le risque d'explosion combinatoire (si l'on veut conserver en mémoire toutes les détections partielles en cours ou si l'IDS est lui-même cible d'une attaque par suffocation [**?**]).

L'approche par scénarios dispose toutefois d'un avantage conséquent : elle permet d'identifier et de nommer le type d'attaque qui a mené à la détection, ce qui n'est pas le cas de l'approche comportementale. Cependant, elle ne permet pas de détecter de nouvelles attaques (contrairement à l'approche comportementale) car on ne dispose pas de leurs signatures. Ces deux caractéristiques font que les approches par scénarios et comportementale sont complémentaires.

Annexe A

Lexique

Ce lexique précise le sens que nous donnons à de nombreux termes utilisés dans ce mémoire. Il est largement inspiré des définitions proposées par l'IDWG (*Intrusion Detection Working Group* de l'IETF.

Administrateur : personne chargée de mettre en place la *politique de sécurité*, et par conséquent, de déployer et configurer les *IDS*.

Alerte : message formaté qui décrit un événement relatif à une action qui compromet la sécurité d'un système ou d'un réseau. Les alertes sont produites par un *analyseur*. Nous faisons l'hypothèse que le format utilisé est l'IDMEF.

Analyseur : outil logiciel qui met en œuvre l'approche choisie pour la détection (comportementale ou par *scénarios*). Il génère des *alertes* lorsqu'il détecte une *intrusion* à partir des *événements* remontés par les *capteurs* ou à partir d'*alertes* générées par d'autres analyseurs.

Approche comportementale : ensemble des techniques utilisées par les *IDS* qui basent leur processus de détection sur l'hypothèse que toute déviation significative du comportement observé d'une entité par rapport à son modèle de comportement normal constitue une *intrusion* potentielle.

Approche par scénarios : ensemble des techniques utilisées par les *IDS* qui détectent les *intrusions* en recherchant dans les activités courantes celles qui sont caractéristiques de *scénarios* d'*attaques* connus (comparaison avec une base de signatures d'*attaques*) . Aussi appelée approche par *signatures*.

Attaque : synonyme d'*intrusion*.

Capteur : logiciel qui génère les *événements* en filtrant et formatant les données brutes intéressantes provenant d'une unique source d'information (paquets du réseau, logs du système ou logs applicatifs).

Détection d'intrusions : processus logiciel de recherche des *intrusions* qui s'appuie sur la surveillance des activités des entités dans les systèmes et

les réseaux. Nous réservons le terme de détection d'intrusions à l'analyse logicielle et automatique par opposition à l'analyse manuelle de logs effectuée par un opérateur humain.

Evénement : message formaté renvoyé par un *capteur*. C'est l'unité élémentaire utilisée pour représenter une étape d'un *scénario* d'*attaque* connu.

Exploit : terme utilisé pour désigner un programme d'attaque.

Faux positif : *alerte* émise en présence d'une action légitime rapportée à tort comme étant une *intrusion* par un *système de détection d'intrusions* (fausse alerte).

Faux négatif : absence d'*alerte* en présence d'une action qui constitue bien une *intrusion* mais qui n'a pas été détectée comme telle par un *système de détection d'intrusions*.

IDS : acronyme de "Intrusion Detection System". Voir *système de détection d'intrusions*.

Intrusion : action (ou tentative d'action) qui a pour conséquence de compromettre l'intégrité, la confidentialité ou la disponibilité d'une ressource (violation de la *politique de sécurité*).

Manager : composant d'un *IDS* permettant à l'*opérateur* de gérer les autres composants. Ses fonctions comportent généralement la configuration des capteurs et analyseurs, la notification des alertes à l'opérateur et éventuellement la réaction.

Opérateur : personne chargée de l'utilisation du *manager* associé à l'*IDS*. Elle propose ou décide de la *réaction* à apporter en cas d'alerte. C'est parfois la même personne que l'*administrateur*.

Politique de sécurité : spécification des règles à respecter dans le réseau d'une organisation, afin de garantir l'intégrité,la confidentialité et la disponibilité des ressources sensibles. Elle définit quelles activités sont autorisées et lesquelles sont interdites.

Réaction : mesures passives ou actives qui peuvent être prises en réponse à la détection d'une attaque, pour la stopper ou pour corriger ses effets.

Scénario : suite des étapes d'une intrusion.

Signature : règle utilisée par certains analyseurs pour identifier parmi les activités surveillées celles qui sont caractéristiques d'une intrusion.

Sonde : regroupement (logique ou fonctionnel) d'un capteur et d'un analyseur.

Source de données : dispositif générant de l'information sur les activités des entités du système d'information (ex : système d'audit d'un système d'exploitation, *sniffer* réseau, ...).

Système de détection d'intrusions : ensemble complet composé de *capteur(s)*, d'*analyseur(s)* et de *manager(s)*.

Annexe B

DTD du squelette d'une description d'attaque

Une description en ADeLe est un document XML valide et conforme à la DTD suivante.

```
<!-- DTD Specification squelette Adele -->
<!ELEMENT ADELE     (EXPLOIT, DETECTION, RESPONSE)>
<!ATTLIST ADELE name   CDATA #REQUIRED
                params CDATA "">
<!-- partie Exploit -->
<!ELEMENT EXPLOIT   (PRECOND, ATTACK, POSTCOND)>
<!ELEMENT PRECOND   (#PCDATA)>
<!ELEMENT ATTACK    (TEXT?, CODE*)>
<!ELEMENT TEXT      (#PCDATA)>
<!ELEMENT CODE      (#PCDATA)>
<!ATTLIST CODE language CDATA #REQUIRED
               filename CDATA "">
<!ELEMENT POSTCOND  (#PCDATA)>
<!-- partie Detection -->
<!ELEMENT DETECTION (DETECT, CONFIRM, REPORT)>
<!ELEMENT DETECT    (EVENTS, ENCHAIN, CONTEXT)>
<!ELEMENT EVENTS    (#PCDATA)>
<!ELEMENT ENCHAIN   (#PCDATA)>
<!ELEMENT CONTEXT   (#PCDATA)>
<!ELEMENT CONFIRM   (#PCDATA)>
<!ELEMENT REPORT    (#PCDATA)>
!-- partie Response -->
<!ELEMENT RESPONSE  (#PCDATA)>
```

Annexe C

Grammaire complète du langage ADeLe

La grammaire détaillée du langage ADeLe qui suit utilise la notation EBNF de l'outil ANTLR qui est un générateur d'analyseur lexical et syntaxique dont la particularité est d'être LL(K). Elle intègre à la fois la reconnaissance du squelette d'une description en ADeLe (document XML, cf annexe B) et celle du contenu de chacune des parties de la description (les informations concrètes sur l'attaque).

Les conventions de notation sont les suivantes :

- tout non-terminal est en minuscules, tout terminal est soit en majuscules, soit encadré par "" ou ".
- (...)0/1 devient (...) ?
- (...)0/n devient (...)*
- (...)1/n devient (...)+
- les alternatives sont séparées par le symbole | (éventuellement groupées entre parenthèses).

```
description: "<?xml version=\"1.0\"?>"
             "<!DOCTYPE ADELE SYSTEM \"specadele.dtd\">"
             "<ADELE" "name"   "=" STRING
                     ("params" "=" "\"" (liste_param)? "\"" )? ">"
              body
             "</ADELE>"
;
liste_param :
             param ("," param)*
;
```

```
body  :      exploit
             detection
             response
;
param  :    ("IN" | "OUT")  TYPE (ARRAY)? VARNAME
;
exploit :   "<EXPLOIT>"
             precondition
             attaque
             postcondition
            "</EXPLOIT>"
;
detection : "<DETECTION>"
             detect
             confirm
             report
            "</DETECTION>"
;
response  : "<RESPONSE>"
             (TEXTE)?
            "</RESPONSE>"
;
precondition :
            "<PRECOND>"
             ( ID "==" STRING ("|" STRING)* )*
            "</PRECOND>"
;
attaque :   "<ATTACK>"
             (text)?
             (code)*
            "</ATTACK>"
;
text    :   "<TEXT>"
             (TEXTE)?
            "</TEXT>"
;
code    :   "<CODE" "language" "=" STRING
                    ("filename" "=" STRING)? ">"
             (TEXTE)?
            "</CODE>"
;
```

```
postcondition :
            "<POSTCOND>"
             (ID ":=" STRING)*
            "</POSTCOND>"
;
detect  :   "<DETECT>"
             events
             enchain
             context
            "</DETECT>"
;
confirm   : "<CONFIRM>"
             (TEXTE)?
            "</CONFIRM>"
;
report  :   "<REPORT>"
             (FIELD ":=" ( STRING
                          |FUNCTION
                          |(FIELD ("|" FIELD)*
                         )
             )+
            "</REPORT>"
;
events :    "<EVENTS>"
             (event)+
            "</EVENTS>"
;
event : ID ":" ID "{" ( FIELD operator FIELD
                      | FIELD (operator | "MATCHES") STRING ) ";" )+
          "}" ID ("," ID)* ";"
;
operator : ( "==" | "!=" | ">" | "<" | ">=" | "<=")
;
enchain :   "<ENCHAIN>"
             scenario
     (timeout)?
     (mindelay | maxdelay)*
            "</ENCHAIN>"
;
scenario : item
         | suite
```

```
;
item     : ID
         | oneamong
         | nonordered
         | without
;
suite   : "("
            ( item ( ";" ( item
                         | ID "^" NUMBER
                         )
                   )+
            | ID "^" NUMBER ( ";" ( item
                                  | ID "^" NUMBER
                                  )
                              )*
            )
  ")"
;
timeout  :  "Timeout" "(" NUMBER ")"
;
mindelay :  "Mindelay" "(" ID "," ID ")" "==" NUMBER
;
maxdelay :  "Maxdelay" "(" ID "," ID ")" "==" NUMBER
;
oneamong  :   "One_among" "{" scenario ("," scenario)+ "}"
;
nonordered  : "Non_ordered" "{" scenario ("," scenario)+ "}"
;
without :  "[" ( suite
               | nonordered
               | without
       )
           "]" "Without" "{" scenario "}"
;
context :   "<CONTEXT>"
             (constraint)*
            "</CONTEXT>"
;
constraint : ID ":=" FIELD
           | FIELD "==" ID
           | FIELD "==" ( FIELD ("==" FIELD)*
```

```
                    | STRING
                    )
            | FIELD ( "!="
                    | ">"
                    | "<"
                    | ">="
                    | "<="
                    )
                          ( FIELD
                          | STRING
                          )
            | FIELD
                   ( "MATCHES" STRING
                   | "IN" "[" STRING ("," STRING)* "]"
                   )
;
```

Annexe D

Grammaire pour le format interne d'un événement de type paquet réseau

Nous spécifions ici le format **interne** d'un événement provenant du réseau. Le plus basique est un événement que l'on peut déterminer en n'examinant qu'un seul paquet réseau. Eventuellement, plusieurs paquets, issus de la fragmentation d'un seul paquet, peuvent être réassemblés. Pour d'autres types d'activités réseau intéressantes du point de vue de la sécurité (scan de ports, établissement d'une connexion TCP, déni de service par *SYN flooding*), il faut mettre en relation plusieurs paquets pour établir le diagnostic : c'est là le rôle d'un analyseur et non plus d'un capteur. Un capteur se contente de filtrer et formater des données brutes mais n'effectue aucun autre calcul «intelligent». Toutefois, ce format interne peut être utilisé par les sondes pour incorporer (lorsque c'est utile) le détail des paquets qui ont déclenché l'alerte (par exemple dans le champ `<AdditionalData>` pour les alertes au format IDMEF).

La grammaire suivante, en notation EBNF, vise à proposer un format pour la représentation sous forme XML d'un paquet réseau (limité ici à une trame Ethernet) lorsqu'il est considéré comme un événement intéressant. Nous ne représentons ici que la partie qui décrit les données brutes qui seront incluses dans l'événement. La partie commune à tous les types d'événement (réseau, système ou applicatif) qui encapsule les données brutes est définie par une DTD à l'annexe E.

On peut se poser la question de l'opportunité d'utiliser XML (format texte plutôt verbeux) à la place d'un format binaire (généralement compact) pour transmettre un événement de type paquet réseau. Cela ne constitue en aucun cas une obligation. Cependant, l'avantage de cette représentation sous forme décodée vient de ce que les balises XML donnent directement une sémantique aux

valeurs qu'ils contiennent. De plus, même si on n'utilise pas au final ce format pour la transmission de l'événement, on peut, dans les signatures de détection, utiliser sa spécification pour accéder de manière formelle aux valeurs contenues dans l'événement.

La décomposition hiérarchique provient directement du modèle en couches TCP/IP où chaque protocole encapsule un protocole de plus haut-niveau. Dans la grammaire que nous proposons, il y a deux types de champs. Le premier type concerne les champs XML qui contiennent directement une valeur. Le second type concerne les balises qui contiennent d'autres balises ; elles ne sont que des indications d'ordre sémantique mais elles permettent de séparer les protocoles des différents niveaux.

Nous avons choisi de ne traiter ici que quelques protocoles réseaux parmi les plus courants dans un but d'exemple. L'extension de la grammaire pour inclure le décodage de nouveaux protocoles ne comporte pas de difficultés majeures. Il faut d'abord consulter leurs spécifications dans les RFC correspondantes et en déduire les noms des nouvelles balises XML à utiliser ainsi que le format (taille) des données. Nous aurions ainsi pu ajouter la spécification de quelques autres protocoles connus tels que : FTP[1], DNS, SNMP, SMTP ainsi que NFS (inclus dans RPC sur TCP/UDP).

Les quelques attributs XML que l'on peut rencontrer dans la grammaire ne constituent que des informations supplémentaires, ie leurs valeurs ne sont pas présentes dans le paquet. Ainsi, l'attribut "time" de la balise "packet" désigne l'horodatage de la capture[2] du paquet réseau et les attributs "length" (des balises "options" et "data") désignent le nombre d'octets du champ correspondant (2 chiffres hexadécimaux par octet).

Les termes en majuscule (MAC, U4, U8, U16, U32 et RAWDATA) désignent des tokens du langage. Ils permettent de préciser le format numérique des données. Le token MAC représente une adresse physique de niveau Ethernet et comporte 6 octets au format hexadécimal séparés par ' :' (par exemple `00:50:ba:a1:78:cc`). Les tokens U`x` (où `x` = 4,8,16,32) représentent des entiers non-signés codés sur `x` bits (tels quels en notation décimale, précédés de '0x' en notation hexadécimale). Le token RAWDATA désigne des données brutes non décodées.

Le choix d'une grammaire détaillée en EBNF (plutôt qu'une DTD) pour spécifier un format en XML tient ici à la possibilité de représenter plus finement le type des valeurs contenues dans les balises et les attributs (contrairement aux types #CDATA et #PCDATA). Toutefois, on peut trouver la DTD correspondante (mais moins détaillée) incluse dans la DTD globale des événements (cf annexe E).

[1] on peut également utiliser les informations tirées du fichier **xferlog** (log applicatif).

[2] information fournie par le capteur. Ex :Libpcap

```
packet    : "<packet time =\"" U32 "." U32 "\">"
               ethernet
            "</packet>";

ethernet  : "<ether>" etherhdr etherdata (ethertrailer)? "</ether>";

etherhdr  : etherdst ethersrc etherproto;

etherdst  : "<dst>"   MAC "</dst>";
ethersrc  : "<src>"   MAC "</src>";
etherproto: "<proto>" U16 "</proto>";

etherdata : (arp | ip);

ethertrailer : "<trailer length=\"" U32 "\">" RAWDATA "</trailer>";

arp       : "<arp>" arphdr arpdata "</arp>";

arphdr    : arphrd arppro arphlen arpplen arpop;

arphrd    : "<hrd>" U16 "</hrd>";
arppro    : "<pro>" U16 "</pro>";
arphlen   : "<hlen>" U8 "</hlen>";
arpplen   : "<plen>" U8 "</plen>";
arpop     : "<op>"  U16 "</op>";

arpdata   : "<arpdata>" arpshw arpsip arpdhw arpdip "</arpdata>";

arpshw    : "<shw>" MAC "</shw>";
arpsip    : "<sip>" U32 "</sip>";
arpdhw    : "<dhw>" MAC "</dhw>";
arpdip    : "<dip>" U32 "</dip>";

ip        : "<ip>" iphdr (ipdata)? "</ip>";

iphdr     : ipver iphlen iptos iplen ipid ipfrag ipttl ipproto ipcheck
            ipsaddr ipdaddr (ipoptions)?;

ipver     : "<ver>"   U4  "</ver>";
iphlen    : "<hlen>"  U4  "</hlen>";
iptos     : "<tos>"   U8  "</tos>";
iplen     : "<len>"   U16 "</len>";
ipid      : "<id>"    U16 "</id>";
ipfrag    : "<frag>"  U16 "</frag>";
```

```
ipttl     : "<ttl>"   U8  "</ttl>";
ipproto   : "<proto>" U8  "</proto>";
ipcheck   : "<check>" U16 "</check>";
ipsaddr   : "<src>"   U32 "</src>";
ipdaddr   : "<dst>"   U32 "</dst>";
ipoptions : "<options length=\"" U32 "\">" RAWDATA "</options>";

ipdata    : (tcp | udp | icmp | igmp)
          | "<data length=\"" U32 "\">" RAWDATA "</data>";

tcp       : "<tcp>" tcphdr (tcpdata)? "</tcp>";

tcphdr    : tcpsport tcpdport tcpseqnum tcpacknum tcphdrlen tcpflags
            tcpwin tcpcheck tcpurg (tcpoptions)? ;

tcpsport  : "<sport>"  U16 "</sport>";
tcpdport  : "<dport>"  U16 "</dport>";
tcpseqnum : "<seqnum>" U32 "</seqnum>";
tcpacknum : "<acknum>" U32 "</acknum>";
tcphdrlen : "<hdrlen>" U8  "</hdrlen>";
tcpflags  : "<flags>"  U8  "</flags>";
tcpwin    : "<win>"    U16 "</win>";
tcpcheck  : "<check>"  U16 "</check>";
tcpurg    : "<urg>"    U16 "</urg>";
tcpoptions: "<options length=\"" U32 "\">" RAWDATA "</options>";

tcpdata   : rpc
          | "<data length=\"" U32 "\">" RAWDATA "</data>";

udp       : "<udp>" udphdr (udpdata)? "</udp>";

udphdr    : udpsrc udpdst udplen udpcheck ;

udpsrc    : "<src>"   U16 "/<src>";
udpdst    : "<dst>"   U16 "/<dst>";
udplen    : "<len>"   U16 "/<len>";
udpcheck  : "<check>" U16 "/<check>";

udpdata   : rpc
          | "<data length=\"" U32 "\">" RAWDATA "</data>";

rpc       : "<rpc>" (rpcfraglen)? rpchdr (rpccall | rpcreply)? "</rpc>";

rpcfraglen: "<fraglen>" U32 "</fraglen>";
```

```
rpchdr    : rpcxid rpctypmsg ;
rpcxid    : "<xid>"     U32 "</xid>";
rpctypmsg : "<typmsg>"  U32 "</typmsg>";

rpccall   : "<call>" rpcversion rpcprognum rpcprogver rpcprocnum
                     rpccred rpcverf (rpcdata)?
            "</call>";
rpcreply  : "<reply>" rpcverf (rpcdata)? "</reply>";

rpcversion: "<version>" U32 "</version>";
rpcprognum: "<prognum>" U32 "</prognum>";
rpcprogver: "<progver>" U32 "</progver>";
rpcprocnum: "<procnum>" U32 "</procnum>";

rpccred   : "<credtyp>" U32 "</credtyp>"
            "<credlen>" U32 "</credlen>"
            ("<creddata length=\"" U32 "\">" RAWDATA "</creddata>")? ;

rpcverf   : "<verftyp>" U32 "</verftyp>"
            "<verflen>" U32 "</verflen>"
            ("<verfdata length=\"" U32 "\">" RAWDATA "</verfdata>")? ;

rpcdata   : "<data length=\"" U32 "\">" RAWDATA "</data>";

icmp      : "<icmp>" icmphdr (icmpdata)? "</icmp>" ;

icmphdr   : icmptype icmpcode icmpcheck ;

icmptype  : "<type>"   U8  "</type>";
icmpcode  : "<code>"   U8  "</code>";
icmpcheck : "<check>"  U16 "</check>";

icmpdata  : "<data length=\"" U32 "\">" RAWDATA "</data>";

igmp      : "<igmp>" igmphdr (igmpdata)? "</igmp>";

igmphdr   : igmptype igmpcode igmpcheck igmpgroup;

igmptype  : "<type>"  U8  "</type>";
igmpcode  : "<code>"  U8  "</code>";
igmpcheck : "<check>" U16 "</check>";
igmpgroup : "<group>" U32 "</group>";

igmpdata  : "<data length=\"" U32 "\">" RAWDATA "</data>";
```

Annexe E

DTD pour le format des événements (EVMEF)

Les capteurs sont chargés de la phase de filtrage et de formatage des données brutes. Malheureusement, il existe presque autant de formats de sortie que de types de capteur. Pour exprimer une corrélation entre événements dans nos signatures, nous avons besoin présenter de manière unifiée l'accès aux champs d'un événement.

C'est pourquoi nous proposons ici un format de représentation des événements en XML, appelé EVMEF[1]. Il est clairement inspiré de l'IDMEF [CD03] mais ne conserve que les parties communes nécessaires à la représentation d'événements (qu'ils soient de type réseau, système ou applicatif).

La notation XPATH abrégée (définie en 2.3.1.1) nous permet d'avoir une syntaxe commune pour spécifier l'accès à la valeur d'un champ dans un événement. Toutefois, utiliser concrètement le format EVMEF comme format de sortie d'un capteur n'est pas obligatoire (car cela impliquerait de réécrire une nouvelle version de tous les capteurs existants). Il est possible d'établir une relation d'équivalence entre l'accès à un champ de l'événement au format EVMEF et l'accès au champ correspondant de l'événement au format natif. Cela signifie qu'un module de traduction peut, à l'exécution, accéder à la valeur d'un champ d'un événement en utilisant la notation commune EVMEF (ex : "Network/packet/ether/ip/dst") alors que l'événement est dans un autre format (ex : tcpdump).

La figure E.1 présente la structure d'un événement au format EVMEF.

Dans le format IDMEF, les alertes sont incluses dans une balise englobante de type "IDMEF-Message". Pour notre format d'événement, c'est une balise de type "Event-Message" qui a été choisie. Elle peut représenter indifféremment

[1]acronyme de *EVent Message Exchange Format*

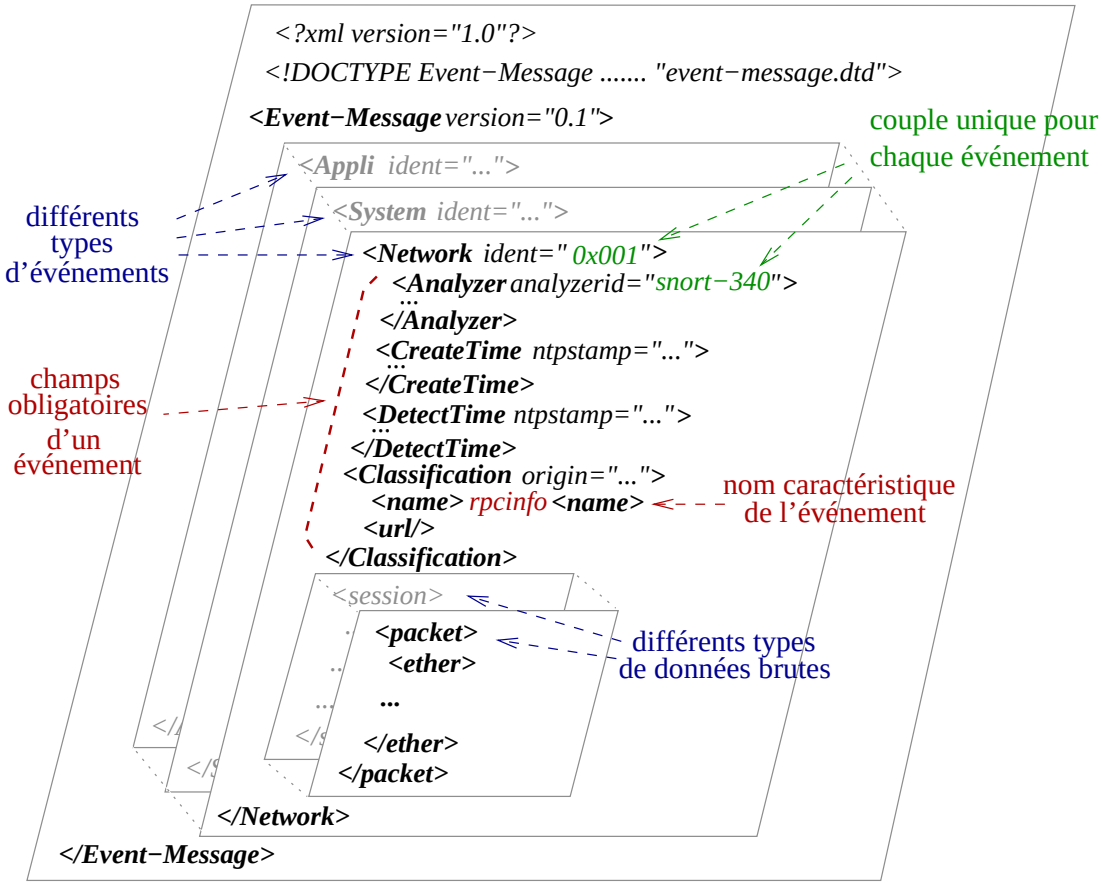

FIG. E.1 – Format EVMEF pour les événements issus de capteurs

un événement de type réseau (<Network>), système (<System>) ou applicatif (<Appli>).

Chaque événement commence par une série de champs obligatoires (`Analyzer`, `CreateTime`, `DetectTime` et `Classification`) dont les spécifications internes sont identiques à celles de l'IDMEF (se reporter à [CD03] pour le détail). Ces champs permettent d'identifier le capteur qui a produit l'événement, de dater sa création et de le classifier (champ `Classification/name`). L'événement se termine par les données brutes qui sont formatées différemment pour chaque catégorie d'événement.

Le format interne des événements est complètement dépendant des données brutes renvoyées par les capteurs, ce qui revient à maintenir un **dictionnaire** du format de chaque type d'événement. Nous avons donc restreint la DTD suivante aux exemples de deux formats d'événements afin de conserver à cette annexe une

taille raisonnable.

On peut remarquer que la spécification sous forme de DTD de ce format d'événements en XML nous oblige à ne définir chaque balise qu'une seule fois de manière globale. Par exemple, la définition de la balise "name" dans le contexte de `Network/Analyzer/Node/name` contraint toute autre occurence de la balise "name" à être du même type (ex : `System/Classification/name`). Une description sous forme de grammaire EBNF (comme celle donnée en annexe D) ne souffre pas de cette limitation. Nous avons fait le choix d'une DTD parce que c'est la représentation traditionnellement utilisée pour spécifier le format des documents XML.

```
<!-- DTD Specification Event-Message
    (System/libsafe & Network/packet) -->

<!ELEMENT Event-Message (System | Network)>
 <!ATTLIST Event-Message version CDATA #REQUIRED>

<!-- evenements de type System -->
<!ELEMENT System (Analyzer, CreateTime, DetectTime, Classification, libsafe)>
 <!ATTLIST System ident CDATA #REQUIRED>

<!-- evenements de type Network -->
<!ELEMENT Network (Analyzer, CreateTime, DetectTime, Classification, packet)>
 <!ATTLIST Network ident CDATA #REQUIRED>

<!-- champs communs aux evenements -->
<!ELEMENT Analyzer (Node,Process)>
 <!ATTLIST Analyzer analyzerid CDATA #REQUIRED>

<!ELEMENT Node (location, name, Address)>
 <!ATTLIST Node category CDATA #REQUIRED>
 <!ELEMENT location (#PCDATA)>
 <!ELEMENT name (#PCDATA)>

<!ELEMENT Address (address)>
 <!ATTLIST Address category CDATA #REQUIRED>
 <!ELEMENT address (#PCDATA)>

<!ELEMENT Process (name)>

<!ELEMENT CreateTime (#PCDATA)>
 <!ATTLIST CreateTime ntpstamp CDATA #REQUIRED>
```

```
<!ELEMENT DetectTime (#PCDATA)>
 <!ATTLIST DetectTime ntpstamp CDATA #REQUIRED>

<!ELEMENT Classification (name,url)>
 <!ATTLIST Classification origin CDATA #REQUIRED>
 <!ELEMENT url (#PCDATA)>

<!-- evenement libsafe -->
<!ELEMENT libsafe (program,message)>
 <!ATTLIST libsafe version CDATA #REQUIRED
                   time CDATA #REQUIRED>
 <!ELEMENT program (name,pid,ppid,uid,gid,euid,egid,stack)>

  <!ELEMENT pid (#PCDATA)>
  <!ELEMENT ppid (#PCDATA)>
  <!ELEMENT uid (#PCDATA)>
  <!ELEMENT gid (#PCDATA)>
  <!ELEMENT euid (#PCDATA)>
  <!ELEMENT egid (#PCDATA)>
  <!ELEMENT stack (addr*)>
   <!ELEMENT addr (#PCDATA)>
 <!ELEMENT message (#PCDATA)>

<!-- evenement packet -->
<!ELEMENT packet (ether)>
 <!ATTLIST packet time CDATA #REQUIRED>

 <!ELEMENT ether (dst, src, proto, (arp | ip), trailer?)>
  <!ELEMENT dst (#PCDATA)>
  <!ELEMENT src (#PCDATA)>
  <!ELEMENT proto (#PCDATA)>
  <!ELEMENT trailer (#PCDATA)>
   <!ATTLIST trailer length CDATA #REQUIRED>

  <!ELEMENT arp (hrd, pro, hlen, plen, op, arpdata)>
   <!ELEMENT hrd (#PCDATA)>
   <!ELEMENT pro (#PCDATA)>
   <!ELEMENT hlen (#PCDATA)>
   <!ELEMENT plen (#PCDATA)>
   <!ELEMENT op (#PCDATA)>
   <!ELEMENT arpdata (shw, sip, dhw, dip)>
    <!ELEMENT shw (#PCDATA)>
    <!ELEMENT sip (#PCDATA)>
```

```
    <!ELEMENT dhw (#PCDATA)>
    <!ELEMENT dip (#PCDATA)>

<!ELEMENT ip (ver, hlen, tos, len, id, frag,
              ttl, proto, check, src, dst,
      options?, (data|tcp|udp|icmp|igmp)? )>
 <!ELEMENT ver (#PCDATA)>
 <!ELEMENT tos (#PCDATA)>
 <!ELEMENT len (#PCDATA)>
 <!ELEMENT id (#PCDATA)>
 <!ELEMENT frag (#PCDATA)>
 <!ELEMENT ttl (#PCDATA)>
 <!ELEMENT check (#PCDATA)>

 <!ELEMENT options (#PCDATA)>
 <!ATTLIST options length CDATA #REQUIRED>

 <!ELEMENT data (#PCDATA)>
  <!ATTLIST data length CDATA #REQUIRED>

 <!ELEMENT tcp (sport, dport, seqnum, acknum, hdrlen, flags,
               win, check, urg, options?, (data | rpc)? )>
  <!ELEMENT sport (#PCDATA)>
  <!ELEMENT dport (#PCDATA)>
  <!ELEMENT seqnum (#PCDATA)>
  <!ELEMENT acknum (#PCDATA)>
  <!ELEMENT hdrlen (#PCDATA)>
  <!ELEMENT flags (#PCDATA)>
  <!ELEMENT win (#PCDATA)>
  <!ELEMENT urg (#PCDATA)>

  <!ELEMENT udp (src, dst, len, check, (data | rpc)?)>

   <!ELEMENT rpc (fraglen?, xid, typmsg, (call|reply)?)>
    <!ELEMENT fraglen (#PCDATA)>
    <!ELEMENT xid (#PCDATA)>
    <!ELEMENT typmsg (#PCDATA)>

    <!ELEMENT call (version, prognum, progver, procnum,
                    credtype, credlen, creddata?,
            verftyp, verflen, verfdata?, data?)>
     <!ELEMENT version (#PCDATA)>
     <!ELEMENT prognum (#PCDATA)>
     <!ELEMENT progver (#PCDATA)>
```

```
    <!ELEMENT procnum (#PCDATA)>
    <!ELEMENT credtype (#PCDATA)>
    <!ELEMENT credlen (#PCDATA)>
    <!ELEMENT creddata (#PCDATA)>
     <!ATTLIST creddata length CDATA #REQUIRED>
    <!ELEMENT verftyp (#PCDATA)>
    <!ELEMENT verflen (#PCDATA)>
    <!ELEMENT verfdata (#PCDATA)>
     <!ATTLIST verfdata length CDATA #REQUIRED>

   <!ELEMENT reply (verftyp, verflen, verfdata?, data?)>

<!ELEMENT icmp (type, code, check, data?)>
 <!ELEMENT type (#PCDATA)>
 <!ELEMENT code  (#PCDATA)>

<!ELEMENT igmp (type, code, check, group, data?)>
 <!ELEMENT group (#PCDATA)>
```

Annexe F

Détails de l'algorithme abstrait utilisé pour la détection

Nous présentons ici l'algorithme abstrait qui définit la sémantique opérationnelle de la détection. Il effectue la reconnaissance d'une signature (modélisée par un automate) dans un flux d'événements.

Nous définissons d'abord les notations utilisées pour les définitions de types abstraits de données et pour le pseudo-langage utilisé par l'algorithme. Nous donnons ensuite le détail des procédures et fonctions qui définissent l'algorithme.

F.1 Notations du pseudo-langage utilisé par l'algorithme

Le pseudo-langage utilisé dans nos algorithmes s'inspire de plusieurs langages de programmation connus :

- C : déclaration de variables ;
- Pascal : définition de types, procédures, fonctions et structures de contrôle ;
- Caml : types n-uplets, unions ;
- Java : accès aux champs d'instances de types structurés, hashtable

Notations

- n-uplet : `(a,b,c)`
- ensemble : `{a,b,c}`
- liste : `[a,b,c]`
- affectation : `:=`
- test d'égalité : `==`
- égal par définition : `=`

Nous utilisons essentiellement trois types de base (entier, chaîne et booléen) ainsi que le type énuméré. Il est aussi possible de définir des types composés de plus haut niveau.

Type énuméré : un type énuméré est défini comme une alternative entre un nombre fini de littéraux, séparés par le caractère '|'.

Exemple :

Sexe = Masculin | Feminin ;
Sexe s := Feminin ;

Type structuré : le type structuré est défini comme une succession de champs nommés pouvant être chacun de type différent. On accède à un champ par la notation pointée (Java). On peut également affecter globalement une variable comme si elle était un n-uplet où chaque élément est affecté au champ qui occupe le même rang dans la définition.

Exemple :

Adresse = (numero : entier, rue : chaine, code : entier, ville : chaine) ;
Adresse adr := (24, "rue des preuves", 35000, "rennes") ;
adr.numero := 25 ;

Type liste : le type liste permet de donner une relation d'ordre sur des éléments de même type. La liste vide se note `"[]"`. On peut connaitre le nombre d'éléments de la liste ($Card$`(<liste>)`). Comme pour un tableau, on autorise un accès direct à l'élément d'indice i ($i = 1..n$) s'il est défini. On peut ajouter un élément en fin de liste (`ajouter(<liste>,<element>)`) et supprimer le dernier élément de la liste (`diminuer(<liste>)`). On permet la copie sélective par valeur d'une tranche de liste (de l'indice min à l'indice max) en utilisant la notation `<liste>[min..max]`. La nouvelle liste obtenue est renumérotée de 1 à (max-min+1).

Exemple :

Agenda = **liste** de chaine ;
Agenda ag := `["lundi"]` ;
ajouter(ag,"mardi") ;
ag[1] := "dimanche" ;
diminuer(ag) ;

Type ensemble : le type ensemble permet de regrouper des éléments de même type, sans ordre particulier et sans doublon. Un ensemble vide se note : $\emptyset$. On ajoute des éléments par l'union ensembliste ("$\cup$") et on les retire par soustraction ("-"). On obtient le cardinal de l'ensemble avec la fonction $Card$`(<ensemble>)`.

Exemple :

Palette = **Ensemble** de chaine ;

Palette pal := {"vert","bleu"} ;
pal := pal ∪ {"rouge"} ;
pal := pal - {"bleu"} ;

Type hashtable : le type hashtable représente une liste de doublets dont le premier élément constitue une clé unique pour accéder au second. A la définition du type, il suffit de donner le type de la clé et le type de la valeur. On considère qu'une hashtable vide est une liste vide ([ ]). La fonction `cles(<hashtable>)` permet d'énumérer toutes les clés (dans l'ordre où elles ont été créées). On autorise l'accès direct à une valeur par la notation `"<hashtable>{<clé>}"`. On peut aussi directement accéder au doublet si l'on connait son rang dans la liste. L'affectation d'une valeur à une clé qui n'était pas présente crée la clé (comme dans le langage Perl). Le retrait d'une clé (et donc de la valeur associée) se fait par `retirer(<hastable>,<clé>)`. Le test de l'existence d'une clé se fait par la fonction booléenne `def(<hashtable>,<clé>)` qui renvoie vrai si la clé est présente et faux sinon.

Exemple :
Table = **hashtable** de (chaine, entier) ;
tab := [ ] ;
tab{"cumul"} := 200 ;
retirer(tab,"cumul") ;
(cle,valeur) := tab[1] ;

N.B. : on considère que tous les passages de paramètres et affectations se font par valeur pour les types de base (entier, chaîne, booléen) et le type énuméré. Ils se font par référence pour tous les types composés que l'on a définis. Comme pour les langages à objets, la création d'une instance se fait grâce au mot clé **new** suivi du nom du type suivi par les paramètres d'initialisation entre parenthèses. Pour les types liste et ensemble, on autorise l'affectation avec des références anonymes (`"[ ...]"` et `"{ ...}"`). Pour réaliser le «clônage» d'une instance de l'un de ces types contenant des références, on utilise l'opérateur `copie(...)` qui descend récursivement dans les structures de données jusqu'aux types de base, en crée des copies distinctes puis recrée les structures de données avec les nouvelles références.

Exemple :
Etat etat := **new** Etat() ;
Automate auto := **new** Automate (∅, ∅, etat, 5) ;
liste de chaine lch1 := ["1", "2"] ;
liste de chaine lch2 := copie(lch1) ;

F.2 Détails de l'algorithme

Les fonctions et procédures suivantes présentent le détail de l'algorithme abstrait utilisé pour définir la sémantique opérationnelle de la détection.

– variables globales –
Automate auto ;
ensemble de (etat : Etat, hypo : Hypothese) kill ;

– algorithme principal –
procedure detection (Log log)
debut
//construction de l'automate
auto := constructionAuto() ;
//marquage initial
initialisationAuto (auto) ;
//analyse du log événement par événement
analyseLog(log) ;
fin

– fonction chargée de construire l'automate correspondant à l'attaque –
fonction constructionAuto ( ) : Automate ;

– procedure chargée d'initialiser l'automate avant de lancer la détection –
procedure initialisationAuto (Automate auto)
debut
// appel à la procédure récursive d'ajout des hypothèses initiales
// cas particulier : l'état initial est son propre prédécesseur !
marquageInitial (auto.init, auto.init, [ ], [ ], [ ], auto.timer) ;
fin

– procédure qui délivre les événements un par un à l'automate –
procedure analyseLog (Log log)
variables
EvenementBrut evtbrut ;
Evenement evt ;
debut
tant que existeElementSuivant(log) **faire**
evtbrut := renvoieElementSuivant(log) ;
evt := filtrage(evtbrut) ;
si (evt.type != "") **alors**

```
            utiliseEvt(evt) ;
        fin si
    fin tant que
fin
```

– fonction qui transforme un événement concret (de type EvenementBrut) en l'événement synthétique correspondant (de type Evenement) qui sera présenté aux transitions de l'automate –
– Si l'attribut type *de l'événement renvoyé est la chaîne vide, alors l'événement concret n'a pas passé le filtrage. –*

```
fonction filtrage (EvenementBrut e ) : Evenement ;
```

– procédure qui simule le traitement d'un événement par l'automate –

```
procedure utiliseEvt (Evenement evt)
debut
    //élimination des hypothèses périmées par Timeout
    destructionTimeout(evt) ;
    //distribution de l'événement aux transitions qui l'attendent
    distribueEvt(evt) ;
    //suppression des hypothèses ajoutées à l'ensemble kill
    supprimeHypotheses() ;
    //ajout des hypothèses nouvellement créées (futur → courant)
    majEtats() ;
fin
```

– procédure récursive chargée de distribuer les hypothèses initiales et de positionner les timer sur les transitions : utilisée à l'initialisation de l'automate et pour démarrer des scénarios inhibiteurs (opérateur Without) –

```
procedure marquageInitial (Etat etat, Etat depuisEtat, liste d'entier sync, liste d'entier with,
    hastable de (chaine,chaine) environ, entier timer)
variables
    Hypothese hypo ;
    entier i ;
    liste d'entier listeSync, listeWith ;
debut
    selon etat.type parmi
    NORMAL : //arrêt de la récursion
        //nouvelle hypothèse vide -> (depuisEtat,sync,with,histo,alias,environ,timeout)
        hypo := new Hypothese(depuisEtat, [0], [0], [ [ ] ], [ ], environ, 0) ;
        //si l'on a franchi plusieurs états de type DISTRIB
        si (Card(sync) != 0 ) alors
```

```
            pour tout i = 1 à Card(sync) faire
                //empilement d'un nouveau numero de synchronisation
                ajouter(hypo.sync, sync[i]) ;
                //empilement d'un nouvel historique local vide
                ajouter(hypo.histo, [ ]) ;
            fin faire
        fin si
        //si l'on a franchi plusieurs états de type WITHOUT
        si (Card(with) != 0 ) alors
            pour tout i = 1 à Card(with) faire
                //empilement d'un nouveau numéro de without
                ajouter(hypo.with, with[i]) ;
            fin faire
        fin si
        //ajout de cette hypothèse initiale
        etat.courant{depuisEtat} := etat.courant{depuisEtat} ∪ {hypo} ;
        //mise à jour du timer de la seule transition sortante
        etat.listeTrans[1].timer := timer ;
    DISTRIB : //imbrication supplémentaire de Non_ ordered
        listeSync := copie(sync) ;
        //ajout d'un nouveau numéro de génération commun
        ajouter(listeSync, nouveauSync()) ;
        //appel récursif vers chacune des branches du Non_ ordered
        pour i := 1 , i < Card(etat.listeTrans)+1 , i++ faire
            marquageInitial(etat.listeTrans[i], etat, listeSync, with, environ, timer) ;
        fin faire
    MULTI : //simple transmission aux états de départ du One_ among
        pour i := 1 , i < Card(etat.listeTrans)+1 , i++ faire
            marquageInitial(etat.listeTrans[i], etat, sync, with, environ, timer) ;
        fin faire
    WITHOUT : //imbrication supplémentaire de Without
        listeWith := copie(with) ;
        //ajout d'un nouveau numéro de génération commun
        ajouter(listeWith, nouveauWith()) ;
        //appel récursif vers la branche positive
        marquageInitial(etat.listeTrans[1], etat, sync, listeWith, environ, timer) ;
    fin selon
fin
```

– fonction qui délivre un numéro de génération différent à chaque appel (pour Non_ ordered) –

```
fonction nouveauSync() : entier ;

– fonction qui délivre un numéro de génération différent à chaque appel (pour
Without) –
fonction nouveauWith() : entier ;

– procédure qui supprime les éventuelles hypothèses périmées (durée de vie dépas-
sée) –
procedure destructionTimeout (Evenement evt)
variables
    Etat etat, depuisEtat ;
    Hypothese hypo ;
debut
    kill := ∅ ;
    pour tout etat ∈ auto.etats faire
        si ( ((etat.type == NORMAL) ou (etat.type == MERGE))
            et etatActif(etat) )
        alors
            pour tout depuisEtat ∈ cles(etat.courant)
                pour tout hypo ∈ etat.courant{depuisEtat} faire
                    si (hypo.timeout != 0) et (evt.heure > hypo.timeout ) alors
                        //marquage des hypothèses à détruire
                        kill := kill ∪ {(etat,hypo)} ;
                    fin si
                fin faire
            fin faire
        fin si
        //on les supprime maintenant
        supprimeHypotheses()
    fin faire
fin

– procédure qui supprime les hypothèses qui sont obsolètes –
procedure supprimeHypotheses()
variables
    Etat e ;
    Hypothese h ;
debut
    pour tout (e,h) ∈ kill faire
        e.courant{h.depuisEtat} := e.courant{h.depuisEtat} - {h} ;
    fin faire
```

```
    //remise à zéro
    kill := ∅ ;
fin
```

– fonction booléenne qui détermine si un état contient des hypothèses –

```
fonction etatActif (Etat etat) : booleen
variables
    Etat depuisEtat ;
debut
    pour tout depuisEtat ∈ cles(etat.courant)
        si (etat.courant{depuisEtat} != [ ]) alors
            retour vrai ;
        fin si
    fin faire
    retour faux ;
fin
```

– procédure qui fournit l'événement à toutes les transitions qui l'attendent –

```
procedure distribueEvt (Evenement evt)
variables
    Transition trans ;
    Hypothese hypo ;
    Etat depuisEtat ;
    entier i, timeout ;
    (pass : booleen, : ensemble de Paire) envvars ;
    hastable de (chaine,chaine) env ;
    liste de Historique histo ;
    liste d'entier sync, with ;
debut
    pour tout trans ∈ auto.transitions faire
        //pour toutes les transitions du bon type qui ont des hypothèses en amont
        si ((trans.type == evt.type) et etatActif(trans.orig)) alors
            //si l'événement passe les contraintes internes
            si (passeContraintesIntra(trans.gardes, evt)) alors
                pour tout depuisEtat ∈ cles(trans.orig.courant) faire
                    //traiter chaque hypothèse
                    pour tout hypo ∈ trans.orig.courant{depuisEtat} faire
                        //cas des transitions qui fixent un timeout aux hypothèses
                        si (trans.timer != 0) alors
                            timeout := evt.heure + trans.timer ;
                        fin si
```

```
                envvars := passeContraintesUnif(trans.gardes, hypo, evt) ;
                //si l'événement passe les contraintes d'environnement
                si (envvars.pass) alors
                    //copie et mise à jour éventuelle des alias
                    si (trans.nomEvt != "") alors
                        alias := copie(hypo.alias) ;
                        alias{trans.nomEvt} := evt ;
                    fin si
                    //copie et mise à jour éventuelle des variables d'environnement
                    env := copie(hypo.env) ;
                    pour tout (nom,valeur) ∈ envvars faire
                        env{nom} := valeur ;
                    fin faire
                    //copie et mise à jour de l'historique
                    histo := copie(hypo.histo) ;
                    //copie de sync et with
                    sync := copie(hypo.sync) ;
                    with := copie(hypo.with) ;
                    //envoi à l'état suivant des informations nécessaires
                        à la création d'1 nouvelle hypothèse
                    ajoutHypothese(trans.dest, sync, with, histo, alias, env, timeout,
                                   trans.orig) ;
                    //transitions qui déclenchent un (des) scénario(s) négatif(s)
                    si (trans.inhib != [ ]) alors
                        pour i := 1 , i < Card(trans.inhib)+1, i++ faire
                            //on passe i niveaux du haut de la pile de with
                            twith := copie(hypo.with[(Card(hypo.with)-i+1)..(Card(hypo.with))]) ;
                            tenv := copie(hypo.env) ;
                            etatNeg := trans.inhib[i] ;
                            marquageInitial(etatNeg, etatNeg, [ ], twith, tenv, timeout) ;
                        fin faire
                    fin si
                sinon
                //l'événement ne passe pas les contraintes d'environnement !
                fin si
            fin faire //hypothèses
        fin faire
    sinon
        //l'événement ne satisfait pas les contraintes internes !
    fin si
fin si
```

fin faire //*transitions*
fin

– fonction booléenne qui détermine si un événement satisfait les gardes intra-événement de la transition –
fonction passeContraintesIntra (**ensemble de Garde** ensGarde, Evenement evt) : booleen ;

– fonction qui détermine si un événement satisfait les gardes inter-événements de la transition au vu de l'hypothèse courante et renvoie éventuellement de nouvelles variables d'environnement –
fonction passeContraintesUnif (**ensemble de Garde** ensGarde, Hypothese h,
Evenement evt) : (pass : booleen, envvars : **ensemble** de Paire) ;

– procédure récursive chargée de traiter/propager une hypothèse qu'une transition a amené dans l'état courant –
procedure ajoutHypothese(Etat etat, **liste** d'entier listeSync, **liste** d'entier listeWith, **liste** d'Historique listeHisto, **hashtable** de (chaine, Evenement) hashAlias, **hashtable** de (chaine, chaine) hashEnv, entier timeout, Etat depuisEtat)
variables
liste d'entier lsync, lwith ;
liste de Hypothese listeH ;
liste de **liste** de Hypothese multiListeH ;
liste d'Etat listeE ;
liste d'Historique lhisto ;
hashtable de (chaine, Evenement) cumulAlias ;
hashtable de (chaine, chaine) cumulEnv ;
liste d'Evenement cumulHisto ;
entier i ;
Etat provenance, etatSuivant ;
Hypothese hypo, h, hypoAlert ;
booleen aucuneListeVide ;
debut
selon etat.type **parmi**
NORMAL : //*création et stockage d'une nouvelle hypothèse*
etat.futur{depuisEtat} := **new** Hypothese(depuisEtat, listeSync, listeWith, listeHisto, hashAlias, hashEnv, timeout) ;
DISTRIB : //*modif et redistribution à chacune des branches du Non_ ordered*
lsync := copie(listeSync) ;
//*ajout d'un nouveau numéro de génération commun*
ajouter(lsync, nouveauSync()) ;
lhisto := copie(listeHisto) ;

```
    //empilement d'un nouvel historique vide
    ajouter(lhisto, [ ]) ;
    //appel récursif vers chacune des branches du Non_ ordered
    pour i := 1 , i < Card(etat.listeTrans)+1 , i++ faire
        ajoutHypothese(etat.listeTrans[i], lsync, listeWith, lhisto, hashAlias, hashEnv,
          timeout, etat) ;
    fin faire
MERGE : //stockage d'une nouvelle hypothèse partielle et tentative de fusion avec celles
        provenant des autres branches
    //création de l'hypothèse
    hypo := new Hypothese(depuisEtat, listeSync, listeWith, listeHisto, hashAlias,
      hashEnv, timeout) ;
    //ajout à l'état courant
    etat.futur{depuisEtat} := hypo ;
    //liste des états de provenance des hypothèses
    listeE := cles(etat.courant) ;
    aucuneListeVide := vrai ;
    multiListeH := [ ] ;
    pour i := 1 , ((i < Card(listeE)+1) && aucuneListeVide) , i++ faire
        //ne pas croiser avec la branche courante
        si ((provenance := listeE[i]) != depuisEtat) alors
            listeH := [ ] ;
            pour tout h ∈ etat.courant{provenance} faire
                si (h.sync[Card(h.sync)] == hypo.sync[Card(hypo.sync)]) alors
                    //ajout si même numéro de génération
                    ajouter(listeH, h) ;
                fin si
            fin faire
            si (Card(listeH) > 0) alors
                ajouter(multiListeH, copie(listeH)) ;
            sinon
                //pas d'hypothèse valide : Non_ ordered non respecté
                aucuneListeVide := faux ;
            fin si
        fin si
    fin faire
    //au moins une hypothèse partielle à fusionner dans chaque branche ?
    si (aucuneListeVide) alors
        //initialisation de l'historique cumulé avec l'historique local de "hypo"
        CumulHisto := copie(hypo.histo[Card(hypo.histo)]) ;
        //idem pour les alias
```

```
        cumulAlias := copie(hypo.alias) ;
        //idem pour les variables d'environnement
        cumulEnv := copie(hypo.env) ;
        etatSuivant := etat.listeTrans[1] ;
        //récursion qui énumère tous les n-uplets et tente une fusion
        //si fusion, alors création d'hypothese(s) et envoi vers l'état suivant par
          ajoutHypothese()
        produitCartesien(etat, CumulHisto, cumulAlias, cumulEnv, multiListeH,
          1, etatSuivant, timeout) ;
    sinon
        //pas de fusion d'hypothèses
    fin si
MULTI : //redistribution sans modification à chacune des branches du One_ among
    pour i := 1 , i < Card(etat.listeTrans)+1 , i++ faire
        ajoutHypothese(etat.listeTrans[i], listeSync, listeWith, listeHisto, hashAlias,
          hashEnv, timeout, etat) ;
    fin faire
MONO : //simple transmission à l'état suivant
    ajoutHypothese(etat.listeTrans[1], listeSync, listeWith, listeHisto, hashAlias, hashEnv,
      timeout, etat) ;
WITHOUT : //transmission d'une hypothèse avec ajout d'un nouveau numéro de
            génération (with)
    lwith := copie(listeWith) ;
    //ajout d'un nouveau numéro de génération commun
    ajouter(lwith, nouveauWith()) ;
    //transmission à la branche positive (seule transition par construction)
    ajoutHypothese(etat.listeTrans[1], listeSync, lwith, listeHisto, hashAlias, hashEnv,
      timeout, etat) ;
SYNC : //réussite/échec d'un opérateur Without (selon branche d'arrivée)
    si (etat.courant[1] == depuisEtat) alors
        //on vient de la branche positive !
        lwith := copie(listeWith) ;
        //on sort d'un Without donc on dépile un niveau
        diminuer(lwith) ;
        //cette hypothèse ne peut plus être invalidée
        ajoutHypothese(etat.listeTrans[1], listeSync, lwith, listeHisto, hashAlias, hashEnv,
          timeout, etat) ;
    sinon
        //on vient de la branche négative !
        //il faut détruire les hypothèses de même génération que cette hypothèse
        echecWithout(etat, listeWith) ;
```

```
        fin si
    FINAL : //création de l'hypothèse et émission d'une alerte
        hypoAlert := new Hypothese(depuisEtat, listeSync, listeWith, listeHisto, hashAlias,
          hashEnv, timeout) ;
        envoiAlerte(etat, hypoAlert) ;
    fin selon
fin
```

– procédure chargée de détruire toutes les hypothèses comportant les mêmes numéros de génération (with) dans le sous-automate positif et dans le sous-automate négatif –

procedure echecWithout (Etat etat, **liste** d'entier listeWith) ;

– procédure récursive qui teste chaque n-uplet d'hypothèses partielles pour les refusionner en une seule –
– Les n-uplets valides ne partagent pas d'événement commun et ont des variables d'environnement compatibles (ie on effectue les tests qui n'ont pas pu être effectués avant sur certaines gardes). Chacun de ces n-uplets donne lieu à création d'une nouvelle hypothèse qui sera envoyée à l'état suivant par ajoutHypothèse() –

procedure produitCartesien(Etat depuisEtat, **liste** d'Evenement cumulHisto, **hashtable** de (chaine, Evenement) cumulAlias, **hashtable** de (chaine, chaine) cumulEnv, **liste** de **liste** d'Hypothese multiListeH, entier niveauRecursion, Etat etatSuivant, entier timeout) ;

– procédure chargée de générer une alerte à partir des informations accumulées par l'hypothèse –

procedure envoiAlerte(Etat depuisEtat, Hypothèse hypo) ;

– procédure chargée d'intégrer les hypothèses nouvellement créées pour prise en compte au pas de simulation suivant –

```
procedure majEtats()
variables
    Etat depuisEtat ;
debut
    pour tout etat ∈ auto.etats faire
        si ( ((etat.type == NORMAL) ou (etat.type == MERGE)) ) alors
            pour tout depuisEtat ∈ cles(etat.futur) faire
                etat.courant{depuisEtat} := etat.courant{depuisEtat} ∪ etat.futur{depuisEtat} ;
                etat.futur{depuisEtat} := [ ] ;
        fin si
    fin faire
fin
```

Annexe G

Deux exemples d'attaques décrites en ADeLe

G.1 Attaque NFS_Mount

La vulnérabilité exploitée dans l'attaque "NFS_Mount" est une configuration incorrecte d'un serveur NFS (*Network File System*) sur un système Unix multi-utilisateurs (de type Solaris dans le cas présent). Lorsqu'un répertoire exporté correspond au répertoire de base d'un utilisateur (*home directory*) et que les droits d'accès à ce partage sont de type lecture-écriture pour tout le monde, il y a alors risque d'intrusion. Le fait que n'importe qui puisse monter à distance le répertoire de l'utilisateur et ajouter (consulter, modifier) les fichiers qui s'y trouvent est déjà quelque chose de dangereux du point de vue de la sécurité. Malheureusement, cela ouvre la voie à une attaque encore plus grave. En effet, si le système concerné permet l'exécution de commandes de type `rlogin`, alors un attaquant va pouvoir usurper l'identité de cet utilisateur en se connectant au système sans connaître le mot de passe.

A l'origine, l'attaquant doit seulement avoir une connectivité réseau avec la machine cible. Il utilise plusieurs commandes (`rpcinfo`, `showmount`, `finger`) afin de collecter des informations. Il sait désormais qu'un serveur NFS tourne sur la machine et que certains répertoires sont exportées en mode lecture-écriture pour tous. Il connait aussi potentiellement des noms de login valides.

S'il trouve un répertoire de base exporté, il crée (sur sa machine locale) un nouveau compte identique à celui de l'utilisateur qui en est propriétaire. Il monte ensuite le répertoire distant dans son arborescence de fichiers par NFS.

Il peut alors créer/modifier le fichier ".rhosts" afin qu'il contienne la chaîne "+ +". Cela a pour conséquence de permettre à n'importe qui (premier '+') depuis n'importe quelle machine (second '+') de pouvoir se connecter sans mot de passe sur cette machine.

L'attaquant peut désormais initier une connexion avec la commande `rlogin` : il a désormais obtenu un niveau d'accès de type "USER" (cf [Ken99]), c'est-à-dire qu'il peut faire tout ce que pouvait faire l'utilisateur dont le compte a été compromis.

Nous décrivons cette attaque en ADeLe de la façon suivante :

```
<?xml version="1.0"?>
<!DOCTYPE ADELE SYSTEM "specadele.dtd">
<ADELE name="NFS_Mount"
       params="IN IPaddr targetip, OUT String account, OUT Connection cnx">

<EXPLOIT>
  <PRECOND>
    Accesslevel == "REMOTE" # niveau d'acces initial requis
  </PRECOND>

  <ATTACK>
    <TEXT>
      Attaque NFS mount
    </TEXT>
    <CODE language="EDL" filename="NFS_Mount.edl">
<![CDATA[
    String output;   # recupere tout de qui est affiche dans la console
    Integer ret_val; # code de retour de la commande
    String rpc_services;
    Integer ret_val0;
    Connection shellhandler;
    EVENT E0{
      #Exec_shell_cmd(<commande_shell>,<affichage_console>,<valeur_retour>)
      Exec_shell_cmd("rpcinfo -p "+targetip,rpc_services,ret_val0);
    }
    IF (ret_val0==0)&&("portmapper" IN rpc_services)&&("mountd" IN rpc_services){
      Non_ordered{ # ordre quelconque!
        [ Integer ret_val1;
          String exported_partitions;
          EVENT E1{
            Exec_shell_cmd("showmount -e "+targetip,exported_partitions,ret_val1);
          }
        ]
        [ Integer ret_val2;
          String users_list;
          EVENT E2{
            Exec_shell_cmd("finger @"+targetip,users_list,ret_val2);
          }
        ]
      }#Non_ordered
      IF (ret_val1==0)&&(ret_val2==0){
        String partition_found;
```

```
        String user;
        #Exists_exported_everyone(<entree>,<liste_partitions>)
        IF Exists_exported_everyone(exported_partitions,partition_found)
        #Cross_part_users(<liste_partitions>,<liste_users>,<user>)
        && Cross_partition_users(partition_found,users_list,user){
          IF !Exists_local_user(user){
            Add_local_user(user); # pas d'evenement observable a distance!
          }
          EVENT E3{
            Exec_shell_cmd("mount -t nfs "+targetip+":/home/"+user+" /home/"+user,
              output,ret_val)}
          }
          One_among{ # ajout de la chaine "+ +" au fichier '.rhosts'
            [ EVENT E4{
                Exec_shell_cmd("echo '+ +' >>~"+user+"/.rhosts",output,ret_val);
              }
            ]
            [ EVENT E5{
                Exec_shell_cmd("echo '+ +' >~"+user+"/.rhosts",output,ret_val);
              }
            ]
          }#One_Among
         EVENT E6{
            shellhandler:=Exec_cmd_shell("rlogin "+targetip+" -l "+account,
            output,ret_val);
          }
          # Maintenant, nous avons le niveau d'acces "USER"
          # mise a jour des parametres de sortie
          account := user;
          cnx := shellhandler;
        }
      }
    }
]]>
    </CODE>
  </ATTACK>

  <POSTCOND>
    AccessLevel        := "USER" # niveau d'acces obtenu
    UnauthorizedResult := "Increased access"
    UnauthorizedResult := "Corruption of information"
  </POSTCOND>
</EXPLOIT>

<DETECTION>
  <DETECT>
    <EVENTS>
        # types d'evenements definis pour cette attaque
      RPCINFO : PACKET { Network/Classification[0]/name == "rpcinfo -p" ;
```

```
      } E0 ;
      SHOWMOUNT : PACKET { Network/Classification[0]/name == "showmount -e" ;
      } E1 ;
      FINGER : PACKET { Network/Classification[0]/name == "finger @" ;
      } E2 ;
      MOUNT : PACKET { Network/Classification[0]/name == "mount home_directory" ;
      } E3 ;
      FAPPEND : BSM { System/Classification[0]/name == "file append" ;
      } E4 ;
      FCREATE : BSM { System/Classification[0]/name == "file create" ;
      } E5 ;
      RLOGIN : PACKET { Network/Classification[0]/name == "rlogin" ;
      } E6 ;
    </EVENTS>
    <ENCHAIN>
       (E0 ; Non_ordered{ E1 , E2 } ; E3 ; One_among{ E4 , E5 } ; E6 )
    </ENCHAIN>
    <CONTEXT>
        # contraintes intra-evenement
      E4/File_Modified/name == ".rhosts"
      E5/File_Created/name == ".rhosts"
        # contraintes inter-evenement
      X := E0/Target[0]/Node/Address/address
      E1/Target[0]/Node/Address/address == X
      E2/Target[0]/Node/Address/address == X
      E3/Target[0]/Node/Address/address == X
      E4/Target[0]/Node/Address/address == X
      E5/Target[0]/Node/Address/address == X
      E6/Target[0]/Node/Address/address == X
    </CONTEXT>
  </DETECT>
  <CONFIRM>
<![CDATA[
    String severity, completion, type ;
      #File_Contains(<fichier>,<contenu>)
    IF File_Contains("/home/"+ E6/Target[0]/User/name +"./rhosts", "+ +"){
      severity := "high";
      completion := "succeeded";
      type := "user";
    } ELSE{
      severity := "medium";
      completion := "failed";
      type := "user";
    }
    Report(severity,completion,type); #variables exportees pour REPORT
]]>
  </CONFIRM>
  <REPORT>
      #construction de l'alerte a partir des valeurs de DETECT et CONFIRM
```

```
    Alert@ident                              := NewAlertid()
    Alert/Analyzer@ident                     := AnalyzerName()
    Alert/Analyzer/Node/Address@category     := "ipv4-addr"
    Alert/Analyzer/Node/Address/address      := AnalyzerIP()
    Alert/CreateTime@ntpstamp                := NewNtpStamp()
    Alert/CreateTime                         := NewTime()
    Alert/DetectTime@ntpstamp                := E6/DetectTime@ntpstamp
    Alert/DetectTime                         := E6/DetectTime
    Alert/Source[0]/Node/Address@category    := "ipv4-addr"
    Alert/Source[0]/Node/Address/address     := E6/Source[0]/Node/Address/address
    Alert/Target[0]/Node/Address@category    := "ipv4-addr"
    Alert/Target[0]/Node/Address/address     := E0/Target[0]/Node/Address/address
    Alert/Classification[0]/origin           := "ADeLe"
    Alert/Classification[0]/name             := "NFS_Mount"
    Alert/Classification[0]/url              := ""
    Alert/Assessment/Impact@severity         := Value(severity,"high")
    Alert/Assessment/Impact@completion       := Value(completion,"succeeded")
    Alert/Assessment/Impact@type             := Value(impact,"user")
  </REPORT>
</DETECTION>

<RESPONSE>
  Reset_TCP_connection( E6/Source[0]/Node/Address/address,
                        E6/Source[0]/Node/Service/port,
                        E6/Target[0]/Node/Address/address,
                        E6/Target[0]/Node/Service/port) ;
</RESPONSE>

</ADELE>
```

Fonctions utilisées dans la description

- `Exists_exported_forall(IN String input, OUT String result)` :
  vrai s'il y a des partitions exportées pour tout le monde dans le premier paramètre ; renvoie leurs noms dans le second paramètre.
- `Cross_part_users(IN String partitions, IN String list_users, OUT String user)` :
  vrai si une partition listée dans le premier paramètre est le répertoire de base[1] de l'un des logins listés dans le second ; renvoie le premier login correspondant dans le dernier paramètre.
- `Exists_local_user(IN String user)` :
  vrai si le login fourni dans le paramètre existe déjà sur la machine de l'attaquant.

[1] nous faisons l'hypothèse que les répertoires de base sont préfixés par `/home`.

Automate de reconnaissance généré pour l'attaque "NFS_Mount" :

```
NAME{NFS_Mount}
INIT{0}
TIMEOUT{0}
STATES(12){
STATE{"0","NORMAL","1","1"}
STATE{"1","DISTRIB","1","2"}
STATE{"2","NORMAL","1","1"}
STATE{"3","NORMAL","1","1"}
STATE{"4","MERGE","2","1"}
STATE{"5","NORMAL","1","1"}
STATE{"6","MULTI","1","2"}
STATE{"7","NORMAL","1","1"}
STATE{"8","NORMAL","1","1"}
STATE{"9","MONO","2","1"}
STATE{"10","NORMAL","1","1"}
STATE{"11","FINAL","1","0"}
}
TRANSITIONS(13){
TRANSITION{"0","1","E0:RPCINFO"}
TRANSITION{"2","4","E1:SHOWMOUNT"}
TRANSITION{"3","4","E2:FINGER"}
TRANSITION{"1","2",""}
TRANSITION{"1","3",""}
TRANSITION{"4","5",""}
TRANSITION{"5","6","E3:MOUNT"}
TRANSITION{"7","9","E4:FAPPEND"}
TRANSITION{"8","9","E5:FCREATE"}
TRANSITION{"6","7",""}
TRANSITION{"6","8",""}
TRANSITION{"9","10",""}
TRANSITION{"10","11","E6:RLOGIN"}
}
CONSTRAINTS(9){
INTRA{E4,==,File_Modified/name,7,.rhosts}
INTRA{E5,==,File_Created/name,7,.rhosts}
UNIF{E0,==,X,Target[0]/Node/Address/address}
UNIF{E1,==,X,Target[0]/Node/Address/address}
UNIF{E2,==,X,Target[0]/Node/Address/address}
UNIF{E3,==,X,Target[0]/Node/Address/address}
UNIF{E4,==,X,Target[0]/Node/Address/address}
UNIF{E5,==,X,Target[0]/Node/Address/address}
UNIF{E6,==,X,Target[0]/Node/Address/address}
}
FILTERS(7){
FILTER(RLOGIN,PACKET){Network/Classification[0]/name == "rlogin"}
FILTER(FAPPEND,BSM){System/Classification[0]/name == "file append"}
FILTER(RPCINFO,PACKET){Network/Classification[0]/name == "rpcinfo -p"}
FILTER(FINGER,PACKET){Network/Classification[0]/name == "finger @"}
FILTER(MOUNT,PACKET){Network/Classification[0]/name == "mount home_directory"}
FILTER(FCREATE,BSM){System/Classification[0]/name == "file create"}
FILTER(SHOWMOUNT,PACKET){Network/Classification[0]/name == "showmount -e"}
}
EXTRACT(7){
FIELDS(RLOGIN,PACKET){Network/Target[0]/Node/Address/address,Network/Source[0]/Node/Address/address,
 Network/DetectTime@ntpstamp,Network/DetectTime}
FIELDS(FAPPEND,BSM){System/Target[0]/Node/Address/address,System/File_Modified/name}
FIELDS(RPCINFO,PACKET){Network/Target[0]/Node/Address/address}
FIELDS(FINGER,PACKET){Network/Target[0]/Node/Address/address}
FIELDS(MOUNT,PACKET){Network/Target[0]/Node/Address/address}
FIELDS(FCREATE,BSM){System/File_Created/name,System/Target[0]/Node/Address/address}
FIELDS(SHOWMOUNT,PACKET){Network/Target[0]/Node/Address/address}
}
```

G.2 Attaque ARP_Spoofing

Sur un réseau local Ethernet, il est possible d'usurper l'identité d'une machine afin d'intercepter le trafic réseau qui lui est destiné. Ce type d'attaque repose sur l'utilisation du protocole ARP. Lorsqu'une machine veut communiquer avec une autre machine du réseau local dont elle connait l'adresse IP, elle effectue une requête ARP pour connaître l'adresse MAC correspondante. La machine possédant cette adresse IP envoie une réponse ARP. Malheureusement, un attaquant du réseau local qui a écouté le trafic réseau peut, à son tour, répondre à la requête en forgeant un paquet ARP contenant sa propre adresse MAC. Dorénavant, l'attaquant reçoit les paquets destinés à l'adresse IP cible car les trames Ethernet émises par la première machine contiennent l'adresse MAC de l'attaquant.

On peut détecter ce type d'attaque en trouvant deux réponses ARP successives qui concernent la même adresse IP mais qui indiquent des adresses MAC différentes dans la réponse. On peut l'exprimer en ADeLe de la façon suivante :

```
<?xml version="1.0"?>
<!DOCTYPE ADELE SYSTEM "specadele.dtd">
<ADELE name="ARP_Spoofing"
       params="">

<EXPLOIT>
  <PRECOND>
    # pour pouvoir écouter les requêtes ARP
    Accesslevel == "LOCAL"
  </PRECOND>

  <ATTACK>
    <TEXT>
      Attaque en ARP spoofing
    </TEXT>
    <CODE language="perl" filename="arpspoof.pl">
      #code
    </CODE>
  </ATTACK>

  <POSTCOND>
    AccessLevel        := "LOCAL" # niveau d'acces obtenu
    UnauthorizedResult := "Corruption of information"
  </POSTCOND>
</EXPLOIT>

<DETECTION>
  <DETECT>
    <EVENTS>
       #filtre sur les paquets ARP de type ARP_REPLY
      ARPREPLY : PACKET { Network/packet/ether/arp/op == "0x0002" ;
```

```
      } A1, A2 ;
    </EVENTS>
    <ENCHAIN>
       (A1 ; A2 )
    </ENCHAIN>
    <CONTEXT>
     #meme @IP declaree pour 2 @MAC differentes
    A1/ether/arp/data/sip == A2/ether/arp/data/sip
    A1/ether/arp/data/shw != A2/ether/arp/data/shw
     #dans un temps tres court (moins d'une seconde)
    Maxdelay(A1,A2) == "1"
    </CONTEXT>
  </DETECT>
  <CONFIRM>
<![CDATA[
   #attention aux eventuels changements d'adresse
   #de clients DHCP => risque de faux positif
]]>
  </CONFIRM>
  <REPORT>
      #construction de l'alerte a partir des valeurs de DETECT et CONFIRM
    Alert@ident                            := NewAlertid()
    Alert/Analyzer@ident                   := AnalyzerName()
    Alert/Analyzer/Node/Address@category   := "ipv4-addr"
    Alert/Analyzer/Node/Address/address    := AnalyzerIP()
    Alert/CreateTime@ntpstamp              := NewNtpStamp()
    Alert/CreateTime                       := NewTime()
    Alert/DetectTime@ntpstamp              := A2/DetectTime@ntpstamp
    Alert/DetectTime                       := A2/DetectTime
    Alert/Target[0]/Node/Address@category := "mac"
    Alert/Target[0]/Node/Address/address   := A2/ether/arp/data/sip
    Alert/Classification[0]/origin         := "ADeLe"
    Alert/Classification[0]/name           := "ARP_Spoofing"
    Alert/Classification[0]/url            := ""
    Alert/Assessment/Impact@severity       := "medium"
    Alert/Assessment/Impact@completion     := "succeeded"
    Alert/Assessment/Impact@type           := "other"
  </REPORT>
</DETECTION>

<RESPONSE>
</RESPONSE>

</ADELE>
```

Annexe H

Exemples d'événements et d'alertes

Les pages suivantes contiennent les exemples d'événements et d'alertes référencés dans la partie mise en œuvre de capteurs et de sondes du chapitre 4.

```
<?xml version="1.0"?>
<!DOCTYPE IDMEF-Message PUBLIC "-//IETF//DTD RFCxxxx IDMEF v0.3//EN" "idmef-message.dtd">
<IDMEF-Message version="0.3">
  <Alert ident="0" impact="unknown">
    <CreateTime ntpstamp="+0000">2001-03-20T18 :29 :28</CreateTime>
    <DetectTime ntpstamp="+0000">2001- 2- 6T14 :25 :11</DetectTime>
    <AnalyzerTime ntpstamp="+0000">collecteur</AnalyzerTime>
    <Analyzer analyzerid="TradGassata">
      <Node category="dns">
        <location>local network</location>
        <name>p34.rennes.supelec.fr</name>
        <Address category="ipv4-addr">
          <address>192.168.0.90</address>
        </Address>
      </Node>
      <Process>
        <name>suntrad_5.6_XML</name>
      </Process>
    </Analyzer>
    <Classification origin="unknown">
      <name>MIR-0160</name>
      <url>?</url>
    </Classification>
    <Source spoofed="no">
      <Node category="dns">
        <location>local network</location>
        <name>p34.rennes.supelec.fr</name>
        <Address category="ipv4-addr">
          <address>192.168.0.90</address>
        </Address>
      </Node>
      <User category="unknown">
        <UserId>
          <name>root</name>
        </UserId>
      </User>
      <Process>
        <name>/etc/security/audit</name>
        <pid>401</pid>
        <path>/etc/security/audit</path>
      </Process>
    </Source>
    <AdditionalData meaning="system" type="string">SOLARIS</AdditionalData>
    <AdditionalData meaning="deamon" type="string">system</AdditionalData>
    <AdditionalData meaning="no" type="string">?</AdditionalData>
    <AdditionalData meaning="event" type="string">AUE_EXECVE</AdditionalData>
    <AdditionalData meaning="return" type="string">failure : Permission denied</AdditionalData>
    <AdditionalData meaning="error" type="string">-1</AdditionalData>
    <AdditionalData meaning="ppid" type="string">401</AdditionalData>
    <AdditionalData meaning="ruid" type="string">root</AdditionalData>
    <AdditionalData meaning="rgid" type="string">other</AdditionalData>
    <AdditionalData meaning="file" type="string">/etc/security/audit</AdditionalData>
  </Alert>
</IDMEF-Message>
```

FIG. H.1 – Evénement délivré au format IDMEF par un capteur système Solaris

```
<?xml version="1.0"?>
<!DOCTYPE Event-Message PUBLIC "-//IETF//DTD RFCxxxx Event v0.1//EN" "event-message.dtd">
<Event-Message version="0.1">
  <System ident="0x00000000">
    <Analyzer analyzerid="LIBSAFE-13777-192.168.0.63">
      <Node category="dns">
        <location>local network</location>
        <name/>
        <Address category="ipv4-addr">
          <address>192.168.0.63</address>
        </Address>
      </Node>
      <Process>
        <name>/usr/sbin/libsafe-sensord</name>
      </Process>
    </Analyzer>
    <CreateTime ntpstamp="0x3ef9c356.0x0">2003-06-25T17:44:22Z</CreateTime>
    <DetectTime ntpstamp="0x3ef9c356.0x0">2003-06-25T17:44:22Z</DetectTime>
    <Classification origin="unknown">
      <name>Libsafe_event</name>
      <url/>
    </Classification>
    <libsafe version="2.0.16" time="1056555862">
      <program>
        <name>/tmp/exploits/t4</name>
        <pid>13779</pid>
        <ppid>13778</ppid>
        <uid>1000</uid>
        <gid>100</gid>
        <euid>1000</euid>
        <egid>100</egid>
        <stack>
          <addr>0x40013ff9</addr>
          <addr>0x40014050</addr>
          <addr>0x40014144</addr>
          <addr>0x8048570</addr>
          <addr>0x80485fd</addr>
          <addr>0x40039a4c</addr>
        </stack>
      </program>
      <message>Overflow caused by strcpy()</message>
    </libsafe>
  </System>
</Event-Message>
```

FIG. H.2 – Evénement délivré au format EVMEF par un capteur système Libsafe

```
<?xml version="1.0"?>
<!DOCTYPE Event-Message PUBLIC "-//IETF//DTD RFCxxxx Event v0.1//EN" "event-message.dtd">
<Event-Message version="0.1">
  <Network ident="0x00000000">
    <Analyzer analyzerid="NETML-1962-192.168.0.63">
      <Node category="dns">
        <location>local network</location>
        <name/>
        <Address category="ipv4-addr">
          <address>192.168.0.63</address>
        </Address>
      </Node>
    </Analyzer>
    <CreateTime ntpstamp="0x3f8bb314.0x0">2003-10-14T10:25:56Z</CreateTime>
    <DetectTime ntpstamp="0x3f8bb314.0x0">2003-10-14T10:25:56Z</DetectTime>
    <Classification origin="unknown">
      <name>rpcinfo</name>
      <url/>
    </Classification>
    <packet time="1066120661.699883">
      <ether>
        <dst>00:48:54:3a:df:fa</dst>
        <src>00:50:ba:a1:78:cc</src>
        <proto>0x0800</proto>
        <ip>
          <ver>4</ver>
          <hlen>20</hlen>
          <tos>0x00</tos>
          <len>96</len>
          <id>4728</id>
          <frag>0x4000</frag>
          <ttl>64</ttl>
          <proto>0x06</proto>
          <check>0xa683</check>
          <src>192.168.0.63</src>
          <dst>192.168.0.13</dst>
          <tcp>
            <sport>35221</sport>
            <dport>111</dport>
            <seqnum>2337374076</seqnum>
            <acknum>2243782422</acknum>
            <hdrlen>32</hdrlen>
            <flags>0x18</flags>
            <win>5840</win>
            <check>0xc688</check>
            <urg>0</urg>
            <options length="12">0101080A00846D0D0B0B76D8</options>
            <rpc>
              <fraglen>0x80000028</fraglen>
              <xid>251826340</xid>
              <typmsg>0</typmsg>
              <call>
                <version>0x00000002</version>
                <prognum>0x000186a0</prognum>
                <progver>0x00000002</progver>
                <procnum>0x00000004</procnum>
                <credtyp>0x00000000</credtyp>
                <credlen>0x00000000</credlen>
                <verftyp>0x00000000</verftyp>
                <verflen>0x00000000</verflen>
              </call>
            </rpc>
          </tcp>
        </ip>
      </ether>
    </packet>
  </Network>
</Event-Message>
```

FIG. H.3 – Evénement délivré au format EVMEF par un capteur réseau

```
<?xml version="1.0"?>
<!DOCTYPE IDMEF-Message PUBLIC "-//IETF//DTD RFCxxxx IDMEF v0.3//EN" "idmef-message.dtd">
<IDMEF-Message version="0.3">
  <Alert ident="2001-12-06T11:46:21Z-21" impact="unknown">
    <Analyzer analyzerid="APACHE_MOD_ID-340-192.168.0.92">
      <Node category="dns">
        <location>local network</location>
        <name/>
        <Address category="ipv4-addr">
          <address>192.168.0.92</address>
        </Address>
      </Node>
      <Process>
        <name>/usr/sbin/apache</name>
      </Process>
    </Analyzer>
    <CreateTime ntpstamp="0x3c0f4c7d.0x0">2001-12-06T11:46:21Z</CreateTime>
    <DetectTime ntpstamp="0x3c0f4c7d.0x0">2001-12-06T11:46:21Z</DetectTime>
    <AnalyzerTime ntpstamp="0x3c0f4c7d.0x0">2001-12-06T11:46:21Z</AnalyzerTime>
    <Source spoofed="unknown">
      <Node category="dns">
        <Address category="ipv4-addr">
          <address>127.0.0.1</address>
        </Address>
      </Node>
    </Source>
    <Target decoy="no">
      <Node category="dns">
        <Address category="ipv4-addr">
          <address>192.168.0.92</address>
        </Address>
      </Node>
      <Process>
        <name>apache</name>
        <pid>340</pid>
        <path>/usr/sbin/</path>
      </Process>
      <Service>
        <name>www</name>
        <port>80</port>
        <protocol>http</protocol>
        <WebService>
          <url>/cgi-bin/phf?Qalias=x%0a/bin/cat%20/etc/passwd</url>
        </WebService>
      </Service>
    </Target>
    <Classification origin="unknown">
      <name>Suspicious URL</name>
      <url/>
    </Classification>
  </Alert>
</IDMEF-Message>
```

FIG. H.4 – Alerte au format IDMEF délivrée par une sonde applicative Apache.

Bibliographie

[ADD00] Almgren (M.), Debar (H.) et Dacier (M.). – A lightweight tool for detecting web server attacks. *In : Proceedings of the Year 2000 Network and Distributed Systems Security Symposium (NDSS 2000).*

[AL01] Almgren (Magnus) et Lindqvist (Ulf). – Application-integrated data collection for security monitoring. *In : Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, éd. par Lee (W.), Mé (L.) et Wespi (A.), pp. 22–36.

[Bis95] Bishop (Matt). – *A standard audit trail format.* – Rapport technique, Department of Computer Science, University of California at Davis, 1995.

[Cal03] Calyx Netsecure. – Netsecure web. – http://www.calyxnetsecure.com/download/NSWeb-4.00/doc-fr/White_Paper.pdf, 2003.

[CD03] Curry (David A.) et Debar (Herve). – Intrusion detection message exchange format data model and extensible markup language (xml) document type definition. – internet draft, January 2003.

[Cis98] Cisco Systems. – Netranger intrusion detection system technical overview. – http ://www.cisco.com/warp/public/778/security/netranger/-ntran_tc.pdf, December 1998.

[CM02] Cuppens (Frédéric) et Miège (Alexandre). – Alert correlation in a cooperative intrusion detection framework. *In : Proceedings of the IEEE Symposium on Security and Privacy.*

[CO00] Cuppens (Frédéric) et Ortalo (Rodolphe). – Lambda : A language to model a database for detection of attacks. *In : Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID'2000)*, éd. par Debar (H.), Mé (L.) et Wu (S. F.), pp. 197–216.

[DBS92] Debar (H.), Becker (M.) et Siboni (D.). – A neural network component for an intrusion detection system. *In : Proceedings of the*

IEEE Symposium of Research in Computer Security and Privacy, pp. 240–250.

[DDMW98] Debar (H.), Dacier (M.), M.Nassehi et Wespi (A.). – Fixed vs. variable-length patterns for detecting suspicious process. *In : Proceedings of the 1998 ESORICS Conference*, éd. par Quisquater (J.J.), Deswarte (Y.), Meadows (C.) et Gollmann (D.), pp. 1–16.

[DDW00] Debar (Hervé), Dacier (Marc) et Wespi (Andreas). – A revised taxonomy for intrusion-detection systems. *Annales des Télécommunications*, vol. 55, n 7-8, 2000.

[Der99] Deraison (Renaud). – *The Nessus Attack Scripting Language Reference Guide*, September 1999.

[EVK00] Eckmann (Steven T.), Vigna (Giovanni) et Kemmerer (Richard A.). – Statl : An attack language for state-based intrusion detection. *In : Proceedings of the ACM Workshop on Intrusion Detection*.

[FHS97] Forrest (S.), Hofmeyr (S.A.) et Somayaji (A.). – Computer immunology. *Communications of the ACM*, vol. 40, n 10, October 1997, pp. 88–96.

[FKP+99] Feiertag (Rich), Kahn (Cliff), Porras (Phil), Schnackenberg (Dan), Staniford-Chen (Stuart) et Tung (Brian). – A common intrusion specification language (cisl). – Specification draft, http ://www.gidos.org/drafts/language.txt, June 1999.

[GD02] Gombault (Sylvain) et Diop (Mamadou). – Fonction de réaction. *In : Proceedings of the NATO 1st Symposium on Real Time Intrusion Detection*.

[Har87] Harel (D.). – Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, vol. 8, 1987, pp. 231–274.

[HCMM92] Habra (Naji), Charlier (Baudouin Le), Mounji (Abdelaziz) et Mathieu (Isabelle). – Asax : Software architecture and rule- based language for universal audit trail analysis. *In : Proceedings of the Second European Symposium on Research in Computer Security (ESORICS'92)*, pp. 435–450.

[HL98] Howard (John D.) et Longstaff (Thomas A.). – *A Common Language for Computer Security Incidents*. – Rapport technique n SAND98-8667, Sandia National Laboratories, October 1998.

[ISO87] ISO. – *ISO 8807 : LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. – Rapport technique, International Standards Organization, February 1987.

[Jen81] Jensen (Kurt). – Coloured petri nets and the invariant-method. *Theoretical Computer Science 14*, 1981, pp. 317–336.

[JLM00] Jacobson (Van), Leres (Craig) et McCanne (Steven). – Tcpdump 3.6 documentation. – http ://www.tcpdump.org, 2000.

[JVL+93] Javitz (H.S.), Valdes (A.), Lunt (T.F.), Tamaru (A.), Tyson (M.) et Lowrance (J.). – *Next Generation Intrusion Detection Expert System (NIDES).* – Rapport technique n A016-Rationales, SRI, 1993.

[Ken99] Kendall (Kristopher). – *A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems.* – Thèse, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[KR02] Ko (Calvin) et Redmond (Timothy). – Noninterference and intrusion detection. *In : Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 177–187.

[KS94] Kumar (S.) et Spafford (E.H.). – A pattern-matching model for misuse intrusion detection. *In : Proceedings of the national computer security conference*, pp. 11–21.

[KS95] Kumar (Sandeep) et Spafford (Eugene H.). – *A Software Architecture to support Misuse Intrusion Detection.* – Rapport technique n CSD-TR-95-009, The COAST Project Department of Computer Sciences, Purdue University, 1995.

[LJ88] Lunt (T.F.) et Jagannathan (R.). – A prototype real-time intrusion-detection expert system. *In : Proceedings of the IEEE Symposium on Security and Privacy*, pp. 59–66.

[LJ01] Lin (Yu) et Jones (Anita). – Application intrusion detection using language library calls. *In : Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001).*

[LMPT98] Lindqvist (Ulf), Moran (Douglas), Porras (Phillip) et Tyson (Mabry). – Designing idle : The intrusion data library enterprise. – Web proceedings of the First International Workshop on Recent Advances in Intrusion Detection (RAID'98), http ://www.raid-symposium.org/raid98, September 1998.

[LTL01] Lin (Yao-Tsung), Tseng (Shian-Shyong) et Lin (Shun-Chieh). – An intrusion detection model based upon intrusion detection markup language (idml). *Journal of Information Science and Engineering*, vol. 17, n 6, November 2001, pp. 899–919.

[LWJ98] Lin (J.-L.), Wang (X.S.) et Jajodia (S.). – Abstraction-based misuse detection : High-level specifications and adaptable strategies. *In : Proceedings of the Eleventh Computer Security Foundations Workshop.*

[MD03] Morin (Benjamin) et Debar (Hervé). – Correlation of intrusion symptoms : an application of chronicles. *In : Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID'03).*

[Mé98] Mé (Ludovic). – Gassata, a genetic algorithm as an alternative tool for security audit trails analysis. – Web proceedings of the First international workshop on the Recent Advances in Intrusion Detection (RAID'98), http ://www.raid-symposium.org/raid98, September 1998.

[MH03] Meier (Michael) et Holz (Thomas). – Intrusion detection systems list and bibliography. – http ://www-rnks.informatik.tu-cottbus.de/en/security/ids.html, January 2003.

[MHL+03] Mell (Peter), Hu (Vincent), Lippmann (Richard), Haines (Josh) et Zissman (Marc). – *An Overview of Issues in Testing Intrusion Detection Systems.* – Rapport technique nNIST-IR-7007, National Institute of Standards and Technology, June 2003.

[MM01] Michel (Cédric) et Mé (Ludovic). – Adele : an attack description language for knowledge-based intrusion detection. *In : Proceedings of the 16th International Conference on Information Security (IFIP/SEC 2001)*, pp. 353–365.

[Mou97] Mounji (Abdelaziz). – *Languages and Tools for Rule-Based Distributed Intrusion Detection.* – Thèse de PhD, Universite de Namur, Juillet 1997 1997.

[MPS78] Moalla (M.), Pulou (J.) et Sifakis (J.). – Synchronized petri nets : a model for the description of non-autonomous systems. *In : Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, pp. 374–383.

[Pax98] Paxson (Vern). – Bro : A system for detecting network intruders in real-time. *In : Proc. of the 7th Usenix Security Symposium.*

[PD00] Pouzol (Jean-Philippe) et Ducassé (Mireille). – Handling generic intrusion signatures is not trivial. – Extended abstract presented at RAID'2000, October 2000.

[PN98] Ptacek (T.H.) et Newsham (T.N.). – Insertion, evasion, and denial of service : Eluding network intrusion detection. – http ://www.securityfocus.com/data/library/ids.ps, January 1998.

[Roe99] Roesch (Martin). – Snort - lightweight intrusion detection for networks. *In : Proceedings of the USENIX LISA'99 conference.*

[RSG02] Reliable Software Group, Department of Computer Science (UCSB). – Alertstat. – http ://www.cs.ucsb.edu/ rsg/STAT/software/index.html, 2002.

[Sec98] Secure Networks. – *Custom Attack Simulation Language (CASL)*, January 1998.

[Sie99] Sielken (Robert S.). – *Application Intrusion Detection*. – Rapport technique nCS-99-17, Dept. of Computer Science, University of Virginia, June 1999.

[Sun] Sun Microsystems. – Sunshield basic security module guide. – Solaris Documentation.

[Sys] syslog (3). – Unix documentation.

[Sys98] Systems (Internet Security). – Network- vs. host-based intrusion detection : A guide to intrusion detection technology. – http ://www.iss.net/prod/nvh_ids/nvh_ids.pdf, October 1998.

[TL00] Templeton (Steven J.) et Levitt (Karl). – A requires/provides model for computer attacks. *In : Proceedings of the 2000 New Security Paradigms Workshop (NSPW'00)*, pp. 31–38.

[TS01] Tsai (Timothy) et Singh (Navjot). – Libsafe 2.0 : Detection of format string vulnerability exploits. – White paper - http ://www.research.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf, February 2001.

[TZ00] Turner (Elliot) et Zachary (Robert). – Securenet pro software's snp-l scripting system. – http ://www.intrusion.com/Downloads/docs/snpl_wp.pdf, July 2000.

[VEK00] Vigna (Giovanni), Eckmann (Steven T.) et Kemmerer (Richard A.). – Attack languages. *In : Proceedings of the IEEE Information Survivability Workshop*.

[VK99] Vigna (Giovanni) et Kemmerer (Richard A.). – Netstat : A network-based intrusion detection system. *Journal of Computer Security*, February 1999.

[VL89] Vaccaro (H.S.) et Liepins (G.E.). – Detection of anomalous computer session activity. *In : Proceedings of the IEEE Symposium on Security and Privacy*.

[VS01] Valdes (Alfonso) et Skinner (Keith). – Probabilistic alert correlation. *In : Proceedings of the Fourth International Symposium on the Recent Advances in Intrusion Detection (RAID'2001)*, éd. par Lee (W.), Mé (L.) et Wespi (A.), pp. 54–68.

[WE02] Wood (Mark) et Erlinger (Michael A.). – Intrusion detection message exchange requirements. – internet draft, October 2002.

[Wor99] World Wide Web Consortium (W3C). – Xml path language (xpath) version 1.0, w3c recommendation. – http ://www.w3.org/TR/xpath, November 1999.

[Wor00] World Wide Web Consortium (W3C). – Extensible markup language (xml), w3c recommendation (second edition). – http ://www.w3.org/TR/REC-xml, October 2000.

[ZMB02] Zimmermann (Jacob), Mé (Ludovic) et Bidan (Christophe). – Introducing reference flow control for intrusion detection at the os level. *In : Proceedings of the 5th International Symposium on the Recent Advances in Intrusion Detection (RAID'2002).*

Résumé

La détection d'intrusions vise à automatiser la détection des actions malicieuses perpétrées sur un réseau de machines par un utilisateur interne ou un attaquant externe. Cela passe par la mise en place d'une surveillance des activités des utilisateurs et des systèmes pour analyse ultérieure de ces activités.

Nous proposons dans cette thèse un langage de haut niveau d'abstraction, ADeLe, dédié à la description des attaques. Ce langage donne les moyens de décrire complètement une attaque sous ses différents aspects : exploit, détection et réaction.

Nous détaillons tout d'abord la syntaxe et la sémantique informelle de ce langage. La partie détection de la description permet de corréler des événements ou des alertes en définissant des signatures qui comportent des contraintes temporelles et logiques.

Nous donnons ensuite la sémantique opérationnelle associée à la partie détection du langage, basée sur des automates à états finis qui modélisent les signatures. Nous présentons un algorithme abstrait qui utilise ces automates pour effectuer la détection. Nous abordons également le problème de l'explosion combinatoire lié à notre modèle qui conserve en cours d'exécution, par défaut, l'ensemble des solutions partielles en mémoire.

Nous présentons enfin différentes réalisations logicielles : plusieurs capteurs système et réseau, une sonde applicative, un compilateur du langage ADeLe vers nos automates à états finis ainsi que l'analyseur ADeLaIDS. Nous présentons également deux expérimentations effectuées à l'aide de cet analyseur.

Mots clés : sécurité des systèmes d'information, détection d'intrusions, langage de description d'attaques, approche par scénarios, signature d'attaque, corrélation d'événements, corrélation d'alertes, automates de reconnaissance.